TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MARKO LEINO

PROCESS SIMULATION UNIT OPERATION MODELS – REVIEW OF OPEN AND HSC CHEMISTRY I/O INTERFACES

Master of Science Thesis

Examiner: Professor Hannu Jaakkola
Examiner and topic approved by the Faculty Council of the Faculty of Business and Built Environment on 6 April 2016

ABSTRACT

MARKO LEINO: Process Simulation Unit Operation Models – Review of Open and HSC Chemistry I/O Interfaces
Tampere University of Technology
Master of Science Thesis, 76 pages, 29 Appendix pages
April 2016
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu Jaakkola

Keywords: CAPE-OPEN, HSC Chemistry Sim, unit operation, process simulation

Chemical process modelling and simulation can be used as a design tool in the development of chemical plants, and is utilized as a means to evaluate different design options. The CAPE-OPEN interface standards were developed to allow the deployment and utilization of process modelling components in any compliant process modelling environment.

This thesis examines the possibilities provided by the CAPE-OPEN interfaces and the .NET framework to develop compliant, cross-platform process modelling components, particularly unit operations. From the software engineering point of view, a unit operation is a representation of physical equipment, and contains the mathematical model of its functionality.

The study indicates that the differences between the CAPE-OPEN standards and Outotec HSC Chemistry Sim are negligible at the conceptual level. On the other hand, at the implementation level, the differences are quite considerable. Regardless of the simulation application being used, the modelling of unit operations requires interdisciplinary skills, and creating tools and methods to ease the development of such models is well justified.

The results of this study suggest that CAPE-OPEN both provides various paths to change the way HSC Chemistry Sim works and offers the HSC development team a chance to determine an alternative way to distribute tasks between simulation components. In addition, making HSC Chemistry Sim compliant would bring benefits, such as an extended process modelling component library, and perhaps more publicity. Obviously, the workload required by the changes depends on the chosen path, which invariably entails a lengthy learning curve. This thesis contributes by helping to make that learning curve shorter.

TIIVISTELMÄ

MARKO LEINO: Process Simulation Unit Operation Models – Review of Open and HSC Chemistry I/O Interfaces
Tampereen teknillinen yliopisto
Diplomityö, 76 sivua, 29 liitesivua
Huhtikuu 2016
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu Jaakkola

Avainsanat: CAPE-OPEN, HSC Chemistry Sim, yksikköoperaatio, prosessien simulointi

Kemiallisten prosessien mallinnusta ja simulointia käytetään kemiallisten tuotantolaitosten suunnittelussa työvälineinä ja niiden avulla voidaan arvioida eri suunnitteluvaihtoehtojen mielekkyyttä. CAPE-OPEN rajapintastandardit on kehitetty mahdollistamaan mallinnuskomponenttien käyttöönotto ja hyödyntäminen missä tahansa yhteensopivassa, standardia noudattavassa, prosessinmallinnusohjelmistossa.

Tässä työssä tutkittiin CAPE-OPEN rajapintojen ja .NET -sovelluskehyksen tarjoamia mahdollisuuksia simulointiohjelmistosta riippumattomien, standardia noudattavien, prosessinmallinnuskomponenttien, erityisesti yksikköoperaatioiden kehittämiseen. Ohjelmistoteknisesti yksikköoperaatio on fyysisen prosessilaitteen kuvaus, joka pitää sisällään kyseisen laitteen toiminnan matemaattisen mallin.

Osoitetuksi tulee, että erot CAPE-OPEN rajapintastandardien ja Outotec HSC Chemistry Sim -simulointisovelluksen välillä ovat käsitteellisellä tasolla pieniä. Toisaalta, toteutustasolla erot ovat suuria. Riippumatta käytettävästä simulointiohjelmistosta yksikköoperaatioiden mallien kehittäminen vaatii lähtökohtaisesti poikkitieteellistä osaamista, ja onkin perustelua kehittää apuvälineitä, joilla yksikköoperaation mallien luominen olisi mahdollisimman vaivatonta.

Työn tulosten perusteella voidaan sanoa, että CAPE-OPEN tarjoaa sekä vaihtoehtoja muuttaa nykyistä HSC Chemistry Simin toimintamallia että kehittäjille mahdollisuuden hahmottaa simulaattorin osien välinen vastuunjako uudessa valossa. Lisäksi, HSC Chemistry Simin muuttaminen yhteensopivaksi standardin kanssa toisi mukanaan etuja, kuten laajennetun komponenttivalikoiman ja mahdollisesti enemmän julkisuutta. Muutoksiin liittyvä työmäärä riippuu valitusta kehityssuunnasta, mutta joka tapauksessa niihin liittyy kestoltaan huomattavan pitkä perehtymisvaihe, jota tämä työ osaltaan auttaa lyhentämään.

PREFACE

This thesis was commissioned by Outotec Finland plc as part of the continuous development work on the Outotec HSC Chemistry software suite. The research was conducted at Outotec Research Center in Pori and at Tampere University of Technology Pori Campus.

During the course of the work I received ample help and support and I would like to express my gratitude for that. Firstly, I would like to thank my instructor Antti Roine, D.Sc. (Tech.), for making it possible for this work to become reality in the first place, for all his help, and for giving me enough leeway to write the content of this study in the way I had envisioned.

I would like to thank Professor Hannu Jaakkola for supervising my thesis work. His professional approach to supervising the work is greatly appreciated. This particular thesis will go down in history as the 302nd thesis he has supervised. On a related note, the other members of the Tampere University of Technology Pori Campus staff are acknowledged for their excellent work and for providing a good learning experience in an open, friendly atmosphere.

In addition, I wish to extend my thanks to the members of the HSC Chemistry development team for their help. Mika Saari, M.Sc. (Tech.), who helped me with software development tools and virtual machines, and Michael Jones, BA (Hons) and Sue Pearson, BA (Hons), from Pelc Southbank Languages, who proofread the thesis, are acknowledged for their help.

Last but not least, I would like to thank my family and friends for their support. Very special thanks go to my wife Rosana and our children Patrick and Daniela for everything.

Pori, Finland, 6 May 2016

Marko Leino

CONTENTS

ABBREVIATIONS AND SYMBOLS

AE	Algebraic equation
API	Application Programming Interface
C#	C# (C sharp) programming language
C++	C++ programming language
CAPE	Computer Aided Process Engineering
CAS	Chemical Abstracts Service
CCW	COM Callable Wrapper
CIDL	CORBA Interface Definition Language
CLSID	Class Identification GUID
CLR	Common Language Runtime
CO-LaN	CAPE-OPEN Laboratories Network
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CTS	Common Type System
DAE	Differential-algebraic equation
DLL	Dynamic-link library
EOS	Equation of state
EPA	[United States] Environmental Protection Agency
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HRESULT	COM error result
IDE	Integrated Development Environment
IDL	Interface Definition Language
IL	Intermediate Language
LP	Linear programming
MIDL	Microsoft Interface Definition Language
MILP	Mixed integer linear programming
MINLP	Mixed integer nonlinear programming
MVC	Model-view-controller architectural pattern
.NET	Microsoft .NET framework
NLAE	Nonlinear algebraic equation

ABBREVIATIONS AND SYMBOLS

NLP	Nonlinear programming
ODE	Ordinary differential equation
OMG	Object Management Group
PDE	Partial differential equation
PEI	Potential environmental impact
PIA	Primary Interop Assembly
PMC	Process Modelling Component
PME	Process Modelling Environment
RCW	Runtime Callable Wrapper
UML	Unified Modelling Language
USEPA	United States Environmental Protection Agency
UUID	Universally Unique Identifier
VB	Visual Basic programming language
VB6	Visual Basic 6 programming language
WAR	Waste Reduction Algorithm

$a(T)$	parameter related to intermolecular forces
b	parameter related to hard-sphere volume
G	total Gibbs energy (J/s)
$\mathcal{J}$	Jacobian matrix
$\mathcal{J}^{-1}$	inverse of the Jacobian matrix
R	ideal gas constant (8.314 460 J/mol/K)
S	total entropy (J/s/K)
T	temperature (K)
V_m	molar volume (m^3/mol)
$\Psi^S_{k,l}$	specific potential environmental impact of chemical k associated with environmental impact category l

1 INTRODUCTION

Steady-state flowsheet simulations are often used as design and analysis tools in the development of chemical process plants. In a flowsheeting environment, unit operation models are joined together by material, energy or information streams in order to close mass and energy balances. Typically, simulation environments come bundled with a built-in set of generic unit operation models, but quite often there is a need for more specialized unit operation models.

CAPE-OPEN interface standards were developed in order to facilitate the exchange of process modelling components between the available simulation environments. Use of the CAPE-OPEN interfaces allows third party unit operations to be used in any compliant process modelling environments. The interface standards have been available since the beginning of the millennium (Pons 2003) and currently all major steady-state flowsheet environments support the CAPE-OPEN standards (van Baten 2009; van Baten and Pons 2014).

Object-oriented programming lends itself to abstraction and component-based designs, and its methods are employed in the development of comprehensive applications such as operating systems (Stallings 2005). Object-oriented designs enable the generation of software parts that are interchangeable, reusable, easily updated, and easily interconnected. In addition, its use leads to better organization of code and paves the way for more extensible and maintainable systems. Consequently, it is used in the development of CAPE-OPEN compliant components.

This thesis examines the objects and interfaces related to CAPE-OPEN unit operations, and endeavours to provide the reader with an overall picture of what is required to implement a fully functioning CAPE-OPEN compliant unit operation. The picture is not limited to unit operations, but is further broadened by introducing aspects such as extensions and add-ins that reveal the versatility empowered by the CAPE-OPEN interfaces. The thesis is intertwined with the development of HSC Chemistry Sim by presenting its proprietary implementation of unit operations, and partly contrasting that against the respective techniques of a standard compliant product. Hence, a comparison of alternative ways to develop unit operations is realized.

The CAPE-OPEN world has benefitted from the contributions made by Jasper van Baten, PhD, and William Barrett, PhD, both of whom have taken part in developing CAPE-OPEN compliant simulation environments (van Baten et al. 2015a; Barrett et al. 2007). Their papers include proposed amendments to the CAPE-OPEN interface set. They have produced useful utilities and add-ins, some of which will be presented in this study.

There are two main chapters in this thesis. The first contains a technology review that focuses on CAPE-OPEN. The second chapter then builds on the preceding technology review, but has its main focus on HSC Chemistry, and also discusses the development of unit operation models for HSC Chemistry as mentioned above. Finally, the results of this study are presented and followed by conclusions concerning the measures that can be taken based on the findings, as well as an assessment of potential limitations to their use.

2 TECHNOLOGY REVIEW

This chapter provides general information on chemical process simulation, and introduces key elements from the frame of reference for this study. However, the focus is on aspects concerning CAPE-OPEN compliant simulation environments, such as program architecture, relevant documentation, requirements and creation of unit operation models, and extensions to the simulation environments. The chapter is concluded with a brief presentation on compliant simulators, which can be regarded as proofs-of-concept in this context.

2.1 An overview of chemical process simulation

Flowsheet simulators are computer applications that are designed to produce representations of industrial processes. Flowsheet simulations contain unit operations representing equipment models, which are connected via material and energy streams. In an overall flowsheet, the unit operation models and the stream models are tied together in a graph of interdependencies, typically represented by a visual diagram. For instance, the flowsheet in Figure 2.1 represents a single stage from an oil and gas separation system. (CO-Lan 2011; van Baten and Pons 2014)

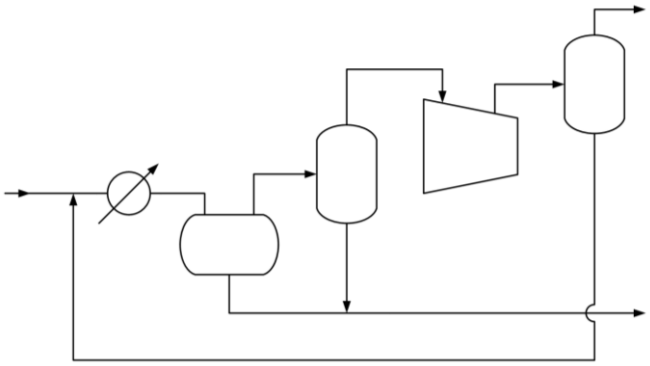


Figure 2.1 Single stage from an oil and gas separation system (adapted from CO-LaN 2011).

A process like the one illustrated in Figure 2.1 is usually entered into a simulator through a graphical user interface (GUI), and in the application window it would look much like the diagram in Figure 2.1. In a real process the unit operations are connected together directly, but in a simulator they are connected to the simulation environment instead of each other. Also, the separation system shown in Figure 2.1 would have three flash separator unit operations in reality, whereas the simulator application would contain only a single flash evaporation algorithm (unit operation) that would be reused as required. (CO-LaN 2011)

Typically, mainstream commercial flowsheet applications contain a number of built-in generic unit operation models, as shown in Figure 2.2. The process models are assembled from a library of existing unit operations. Generally, it is possible to add custom-made unit operations to the unit operation library via a set of interfaces. (CO-LaN 2011; van Baten and Pons 2014)

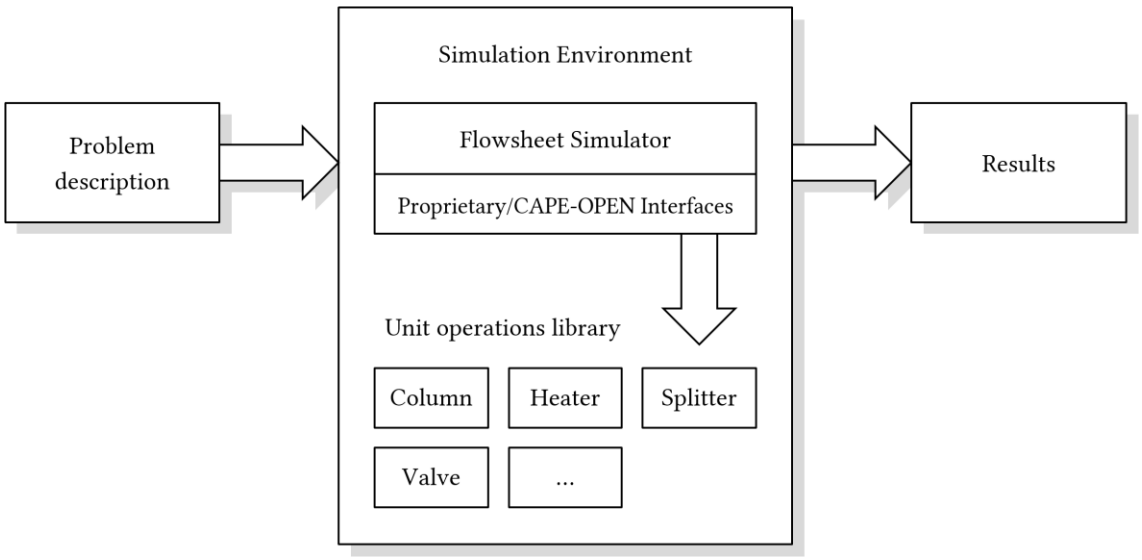


Figure 2.2 Flowsheet simulator schema (adapted from CO-LaN 2011).

In general, a process simulation involves taking the flow rates, compositions and thermodynamic conditions of the input streams, and performing a series of iterative calculations as the streams are processed through unit operations and recycles, see Fig. 2.1 and 2.3. The flowsheet is evaluated to locate recycle loops using a graph theory routine, and the iterative calculations are repeated until user-defined convergence criteria are met, leading to the output stream flow rates and compositions. (Barrett and Yang 2005; Edwards 2013; van Baten and Pons 2014)

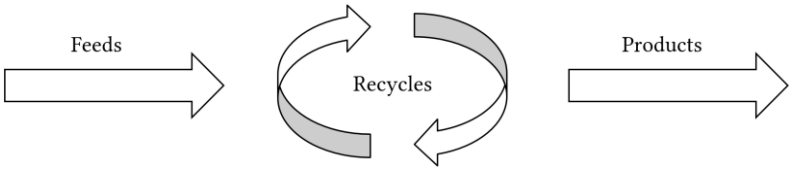


Figure 2.3 Input streams, recycles and output streams (adapted from Edwards 2013).

In addition, when solving a flowsheet, heat and mass balances have to be closed and unspecified process conditions resolved. For example, the law of conservation of mass, which states that mass can neither be created nor destroyed, enforces the requirement that a process design

is not complete until all inputs and outputs of the overall process and individual units satisfy a mass balance for each component. To ensure that the heat and mass balances are calculated in a thermodynamically consistent manner, thermodynamic models are defined at flowsheet level and made available to the unit operations. (Edwards 2013; van Baten and Pons 2014)

There are two main approaches to flowsheet simulation: sequential modular and equation-oriented. Sequential modular is considered powerful and easily accessible to many engineers for the solution of steady-state flowsheet problems. In steady-state models, a mass and energy balance is performed for a stationary process, i.e. for a process that is in an equilibrium state. Steady-state models are simpler and independent of time, as opposed to equation-oriented simulations, which are more complex and largely used for complicated problems, such as dynamic (time-dependent) simulation. Furthermore, batch and semi-batch processes require dynamic simulation whereas continuous processes are studied with both dynamic and steady-state simulators. (Morton 2002; Barrett and Yang 2005; Edwards 2013)

The chart in Figure 2.4 shows the basic steps involved in setting up a steady-state simulation in the ChemCAD process simulator. The chart is read from left to right, and top to bottom.

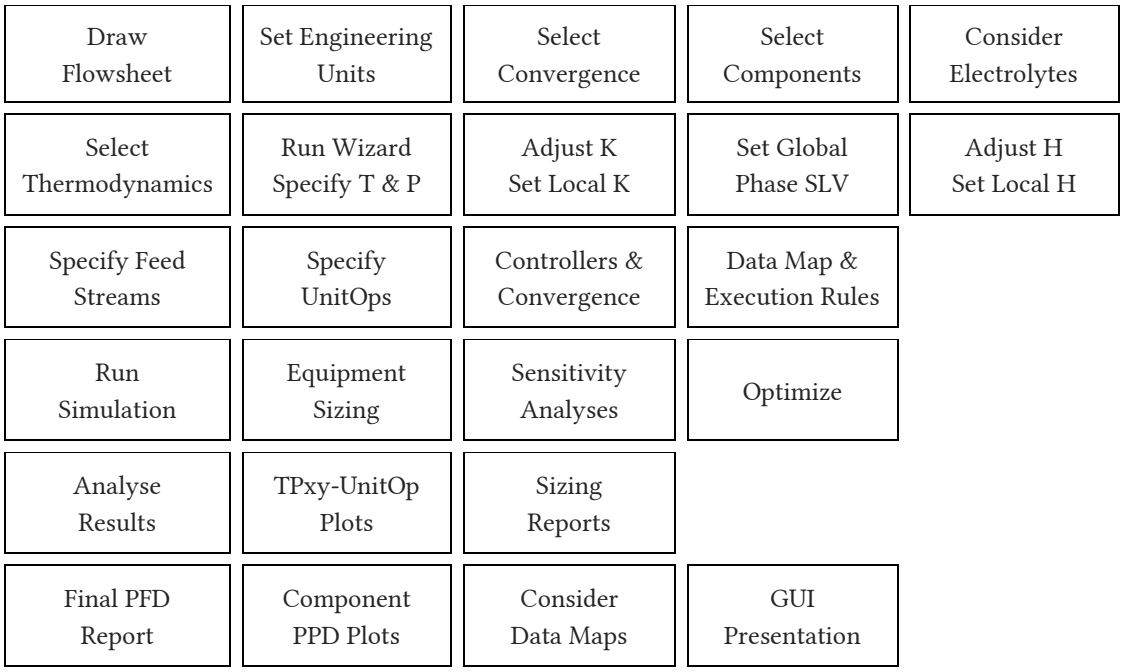


Figure 2.4 Steady-state simulation (adapted from Edwards 2013).

With most simulators the workflow is likely to include most of the steps shown in Figure 2.4, such as the creation of the flowsheet, selecting the unit operations to be used, and entering the feed streams. However, there may be differences regarding other features, such as the plotting capabilities. Also, the selection of available thermodynamic methods depends on the simulator. A brief introduction to the selection of simulation applications is presented in Section 2.4.

2.2 Frame of reference

This section introduces the idea of interoperability between software application and software components. There are some benefits that are achievable only via interoperability across application boundaries. Relevant to the discussion are the application programming interfaces that enable communication between various software components. In order to avoid confusion, a definition of interface is included, along with an introduction to HSC Chemistry Sim.

2.2.1 Interoperability and CAPE-OPEN

Flowsheet applications typically allow the user to deploy custom-made unit operations and thermodynamic models via their proprietary interfaces. This activity usually requires creating the models in a framework specific to the flowsheet application at hand, and then dynamically or statically linking the models to the flowsheet application. Porting the models to other flowsheet applications, or in some cases even to an updated version of the same flowsheet application, requires additional programming effort. (van Baten and Pons 2014)

Making unit operations or thermodynamic models available for multiple flowsheet applications, so-called interoperability (Figure 2.5), involves additional software development work. As a result, custom-made models are made available for use with only a limited number of flowsheet applications, often only one. (van Baten and Pons 2014)

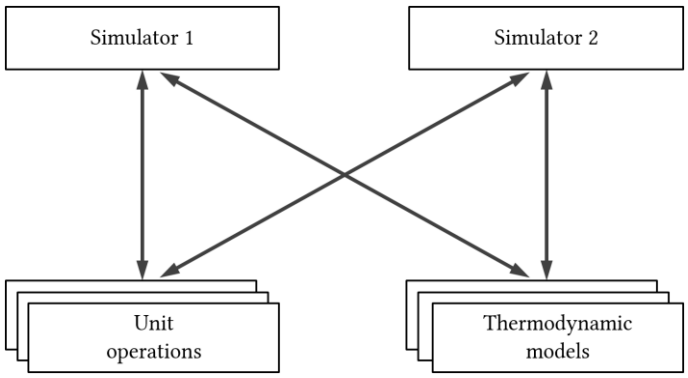


Figure 2.5 Interoperability (adapted from van Baten and Pons 2014).

An industry standard called CAPE-OPEN (Computer Aided Process Engineering OPEN) has been developed to enable interoperability between process simulation software. The CAPE-OPEN standards define a single set of common interfaces for interaction with unit operations and thermodynamics. Essentially, the interface set is designed to be independent of the simulation application or the unit operation model or the thermodynamic model. In terms of software architecture, CAPE-OPEN can be described as plug-and-socket architecture, where the simulation application represents the socket into which the unit operation and thermodynamic models are plugged. Each modelling component, or each socket or plug, is responsible for

providing its own implementation of the CAPE-OPEN interfaces. Therefore, only a single interface implementation has to be developed for each modelling component, and maintained in order to achieve interoperability. The thermodynamic models and unit operation that implement the interfaces can be used in multiple simulation applications. (van Baten and Pons 2014)

The CAPE-OPEN idea originated in the early 1990s with the concept of taking unit operation models from the shelf and plugging them into any process simulator. Two projects, CAPE-OPEN and Global CAPE-OPEN, were jointly funded by industry and the European Community within the Industrial and Materials Technologies Programme. The CAPE-OPEN project ran from January 1997 to June 1999, and resulted in an interim version of the CAPE-OPEN standards. The Global CAPE-OPEN project, which ran from July 1999 to March 2002, further developed the standards. The CAPE-OPEN Laboratories Network, CO-LaN, was founded in 2001 to take care of updating, maintaining and publishing the interface specification. The CO-LaN consortium has companies, software vendors, academic institutions and other parties as its members, with currently almost a hundred members worldwide. (van Baten and Pons 2014)

Generally, it was anticipated at the time of release of the CAPE-OPEN interface specifications that simulator vendors, and other developers, of compliant components would initially to a large extent wrap existing code so that it presents the correct software interface. Concerning any new product developed after the release of the interface specification, the developers are expected to incorporate the standards directly into their designs so that the code will not require the wrapping layer. (CO-LaN 2000)

Another interoperability standard specification that has been applied to process simulation is OPC (www.opcfoundation.org). However, OPC focuses on industrial automation and on access in real time and historical data and events, instead of model interaction between process modelling components like unit operations and thermodynamics. It appears that the only alternative to using CAPE-OPEN is to use the proprietary interfaces of each simulation application. (van Baten and Pons 2014)

With regard to model development, the developers are able to reach out to a wider market by making their products CAPE-OPEN compliant. When compared to a scenario where a unit operation model or a thermodynamic model requires a specific interface for each individual simulation application, the use of CAPE-OPEN interfaces reduces the effort needed for software development and maintenance. Also, it allows a faster access to market and creates a common basis for collaboration in joint projects. Widespread support for CAPE-OPEN benefits commercial model developers, and also research institutes, such as universities, that develop unit operation models or thermodynamic models, as CAPE-OPEN provides a way to disseminate research results to industrial users. (van Baten and Pons 2014)

A company using a particular flowsheet application may wish to switch to another simulation application at some point. The company may have acquired know-how that it has translated into a set of proprietary unit operation models and thermodynamic models. It is of importance then to have the option of transferring the models into the new flowsheet applications. Safe transfer is possible between two CAPE-OPEN compliant applications. Also, interoperability and transferability between software components enables increased flexibility, as the most suitable unit operation and thermodynamic models can be combined with the most suitable simulation application for a particular study. (van Baten and Pons 2014)

2.2.2 Contextual definition of interface

In object-oriented programming, inheritance enables a subclass object to use the functionality of a superclass. There may be cases when one wishes to create a subclass that has the functionality of more than one superclass. One way of accomplishing this would be to allow a subclass to inherit from more than one superclass; for example C++ is a language that allows this kind of multiple inheritance. However, most modern object-oriented languages, such as Java, C# and Visual Basic, limit a class to inheriting from only one superclass. Instead of multiple inheritance, a feature known as interfaces is used to enable a class to adopt some functionality from one class and other functionality from an entirely different class. (Stallings 2005). For terms and their definitions relevant to the discussion, see Table 2.1.

Table 2.1. Object-oriented terms (Stallings 2005)

Term	Definition
Interface	A description closely related to an object class. An interface contains method definitions (without implementations) and constant values. An interface cannot be instantiated as an object.
Method	A procedure that is part of an object and that can be activated from outside the object to perform certain functions.
Object	An abstraction of a real-world entity.
Object Class	A named set of objects that share the same names, sets of attributes, and services.
Object Instance	A specific member of an object class, with values assigned to the attributes.

There is room for confusion, as the term interface is used in much of the literature on objects with both a general-purpose and a specific functional meaning. An interface in the context of designing operating systems, and in that of CAPE-OPEN interfaces, specifies an application-programming interface (API) for certain functionality. An interface does not define any implementation details for the API. The interface definition includes just the method names, the arguments passed, and the type of value returned. An interface may be implemented by a class. Implementing an interface works in much the same way as inheritance. If a class implements an interface, it must have an implementation for all the properties and methods defined in the interface. The implemented methods can be coded in any fashion, as long as the method signature (name, arguments and return type) of each implemented method is identical to the respective definition in the interface. (Stallings 2005)

2.2.3 HSC Chemistry Sim

HSC Chemistry Sim (Sim) is a sequential modular steady-state flowsheeting environment developed by Outotec plc. It is available for purchase as part of the Outotec HSC Chemistry software suite that is designed for performing chemical reaction and equilibrium calculations.

The development of HSC Chemistry (HSC) was initiated in the 1970s and was at the time one of the first software packages designed for chemical, thermodynamic and mineral processing calculations. In fact, the primary reason for the development of HSC was the lack of suitable software for performing such tasks. At the beginning, it was solely used in-house by Outokumpu plc to improve the efficiency of its chemical process research and development. Sales to other companies and academia started in 1987. Over the years, several universities have collaborated in the development of the functionality of HSC Chemistry. (Outotec 2016a)

Outokumpu plc went through organizational changes in 2006 and consequently its technology division was separated from the parent company and named Outokumpu Technology. In 2007, the name was changed to Outotec. The copyrights of the HSC Chemistry suite are currently owned by Outotec and the originator of HSC Chemistry Antti Roine, D.Sc. (Tech.), who have continued to develop HSC Chemistry in co-operation with their collaborators. (Outotec 2016b)

The HSC Chemistry 6.0 software suite, launched in 2006, came bundled with the very first version of Sim. It was designed to run on Windows XP and NT 4.0 operating systems, and its source code was written in Visual Basic 6 (VB6) and C++ programming languages. Sim was improved for the 2009 launch of HSC Chemistry 7.0 with the choice of programming languages unchanged. Prior to the launch of HSC 8.0 in 2014, the HSC development team opted for a major overhaul concerning the development platform. Sim, along with the rest of the HSC suite, was rewritten to work on the .NET Framework, with some bits of code (pun intended) still written in C++, but the majority of source code now written in Visual Basic (see Sec. 3.2.2). The two updates that followed were released in 2015. The latest HSC bundle that has been launched to date – version 9.0 – saw the light of day in late 2015. (Outotec 2016a)

Sim is considered suitable for modelling a variety of processes in the chemical industry, metallurgy, mineralogy, economics and related fields. The processes are simulated in one of four different modes, i.e. Particle, Reaction, Distribution and Other. The Particle mode is designed to deal with processes involving ore and mineral concentrates whereas the Reaction mode is typically used for modelling leaching processes, with dissolution calculations and properties of aqueous solutions having a pivotal role in solving the models. The Distribution mode is designed for modelling pyrometallurgical processes involving unit operations like smelting and combustion furnaces, with (experimental) elemental compositions being utilized in the simulations. (Outotec 2015; Outotec 2016c; Hietala 2013)

The unit operations in Sim are built around spreadsheets. The compositions of feed, product and interim streams as well as the values of state variables like pressure and temperature are defined in the spreadsheets. One of the foci in the development work (Kentala and Lamberg 2011; Remes and Kentala 2014) has been on making Sim user-expandable by allowing users to create their own unit operations deployed in the form of class library assemblies (.dll). However, the way the class library based unit operations interact with Sim does not conform to the CAPE-OPEN standards, and proprietary interfaces are used instead (see Section 3.2.1).

2.3 CAPE-OPEN compliant process simulation environment

This section provides an introduction to the software architecture of CAPE-OPEN compliant simulation environments and gives an overview of the document set that contains the CAPE-OPEN interface specifications. Also, information is presented on the requirements for developing a unit operation that would work in a suitable simulation environment. A case example of creating such a unit operation is given, along with additional information on the related material objects and property packages. Additionally, extensions and add-ins to the interface standard are briefly discussed.

2.3.1 Program architecture

In the context of CAPE-OPEN, the terms process modelling environment (PME) and CAPE-OPEN simulation environment are used to signify simulation applications that have built-in support for CAPE-OPEN. These applications are typically but not exclusively chemical flow-sheet applications. PMEs are the sockets that the process modelling components (PMCs) can be plugged into. Unit operations and thermodynamic packages are the types of PMCs that are widely supported. However, CAPE-OPEN includes definitions for other PMCs too, such as reaction packages. On the whole, CAPE-OPEN merely provides the interfaces that allow software components to communicate with each other, see Fig. 2.6. (van Baten and Pons 2014)

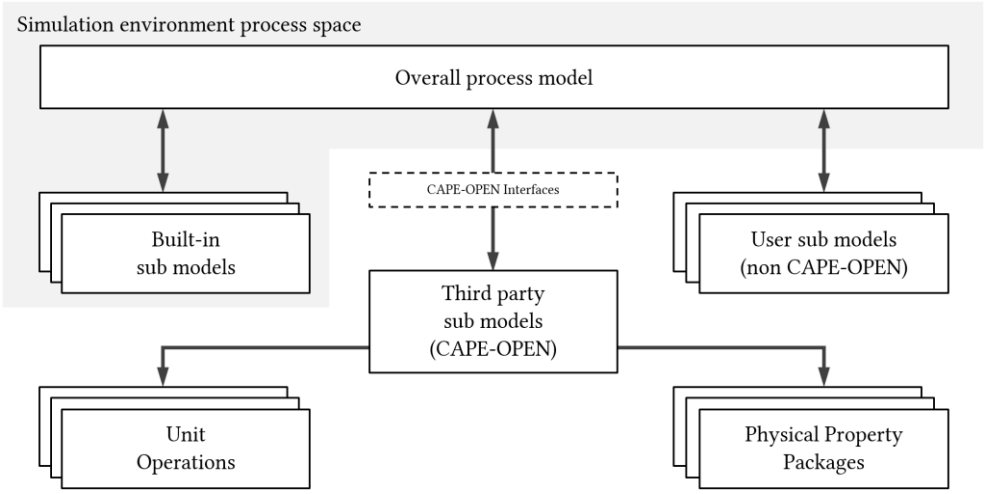


Figure 2.6 Simulation environment (adapted from van Baten and Barrett 2009).

As such, no actual software implementation is associated with CAPE-OPEN. The only binary redistributable part of CAPE-OPEN is the type library file (.tlb) that defines the interfaces. The software components that implement the functionality of the CAPE-OPEN interfaces are the various PMEs and PMCs. Currently, a number of commercial and non-commercial (including open source) implementations of PMCs have been made available (see Section 2.4), in addition

to the many implementations intended exclusively for in-house use. Unit operations and thermodynamic packages are by far the most commonly implemented types of PMCs. The first CAPE-OPEN unit operation that was made public was developed in 2003, soon after the release of the CAPE-OPEN interface specifications. (van Baten and Pons 2014)

Consequently, CAPE-OPEN defines the interfaces that two software components can use to interact. In addition to the interface definitions, an agreement on binary compatibility is required. Such binary compatibility is enabled by middleware (Fig. 2.7). CAPE-OPEN interfaces are provided for two types of middleware, either the Microsoft Common Object Model (COM) or the Common Object Request Broker Architecture (CORBA). (van Baten and Pons 2014)

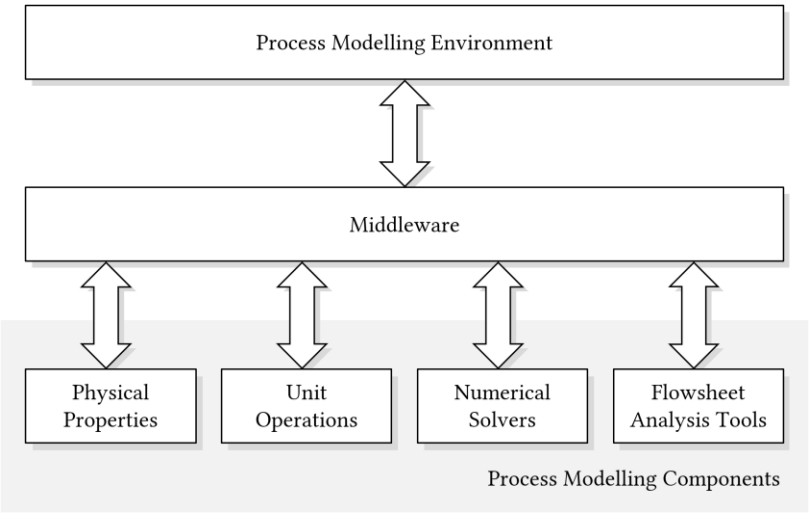


Figure 2.7 CAPE-OPEN components (adapted from Morales-Rodríguez et al. 2008).

Almost all the currently available PMEs and PMCs are based on COM middleware. As COM is an integral part of the Microsoft Windows operating systems, CAPE-OPEN has become somewhat platform-dependent. CO-LaN has identified the problem and is trying to address the issue by, for example, seeking to develop new middleware specific to CAPE-OPEN. Such middleware could be optimized especially for CAPE-OPEN use, which would lead to additional advantages, such as more efficient operation and easier and less error-prone CAPE-OPEN development. (van Baten and Pons 2014)

CAPE-OPEN has also been used as the design basis of PMEs. Three general purpose PMEs are known to have been built with CAPE-OPEN as a basis: COCO, MFFPPT and SolidSim. COCO, developed by AmsterCHEM, is often used as the testing platform for new CAPE-OPEN components. The Metal Finishing Facility Pollution Prevention Tool (MFFPPT), developed by USEPA, supports the CAPE-OPEN interface standards and is suited for general chemical process simulation. SolidSim from AspenTech is designed to be a generally applicable flowsheet application that includes the necessary functionality to simulate solid processes, such as physical treatment, thermal treatment, and reactions. (van Baten and Pons 2014)

The process simulation object model adapted by CAPE-OPEN includes objects to describe unit operations, thermodynamic calculations routines, material streams, energy streams, and generic information streams, as shown in Figure 2.8. Material, energy and information streams contain details about flows within the process, while unit operations represent simulated process equipment. The unit operation objects utilize inlet stream information and input parameters to calculate output stream conditions and parameters. PMEs typically use a flowsheet arrangement to manage the collection of unit operations and streams, providing the user with a graphical representation of the object graph. The PME is in charge of solving the entire flowsheet based upon the thermodynamic and unit operation models provided by the PMCs, which is typically an iterative process. (Barrett et al. 2011)

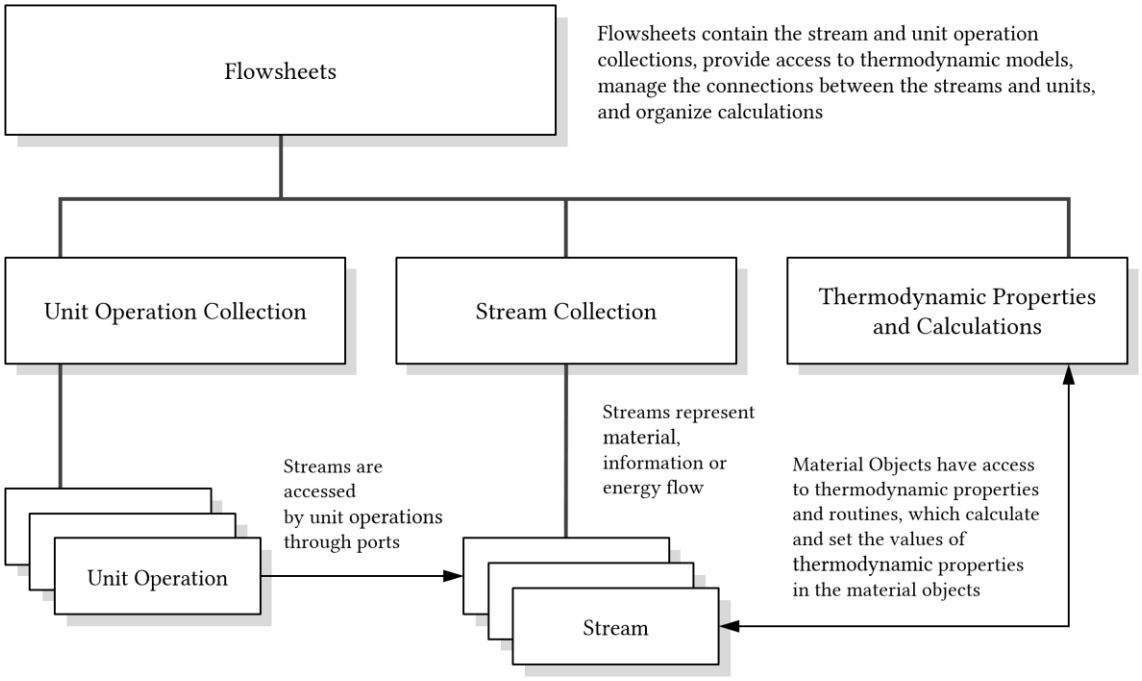


Figure 2.8 Overview of the CAPE-OPEN object model (adapted from Barrett et al. 2011).

The centrepiece in the communication between the PME and the PMCs are the objects that represent material streams. In documents concerning CAPE-OPEN-based component development, these are usually referred to as Material Objects. The communication is enabled by the CAPE-OPEN interfaces that the Material Objects implement. Material Objects are connected to the material ports of unit operation PMCs. Additionally, they serve as property storage objects in communication with Property Package PMCs. (van Baten and Barrett 2009)

Property Packages are thermodynamic calculation engines that calculate thermodynamic and physical properties, thermodynamic phase equilibria, and contain details about the chemical compounds and phases. The Property Packages store calculation results in Material Objects. Besides being a storage for physical and thermodynamic properties, a Material Object keeps track of its current state: which phases are present, and whether the material object is in

thermodynamic phase equilibrium. CAPE-OPEN also specifies special properties for Material Objects, which are pressure, temperature, flow, total flow, fraction and phase fraction. Besides the special properties, other properties can be stored in a Material Object on demand, such as enthalpy, density and viscosity. (van Baten and Barrett 2009)

Figure 2.9 shows the organization of software components. The unit operation and property package PMCs are external components, which are typically, but not necessarily, implemented in the form of COM-compatible dynamic-link library (DLL) files. Instances of the Material Object are created on demand by the simulation application. The shaded area in Figure 2.9 represents the application space of the simulation application. (van Baten and Barrett 2009)

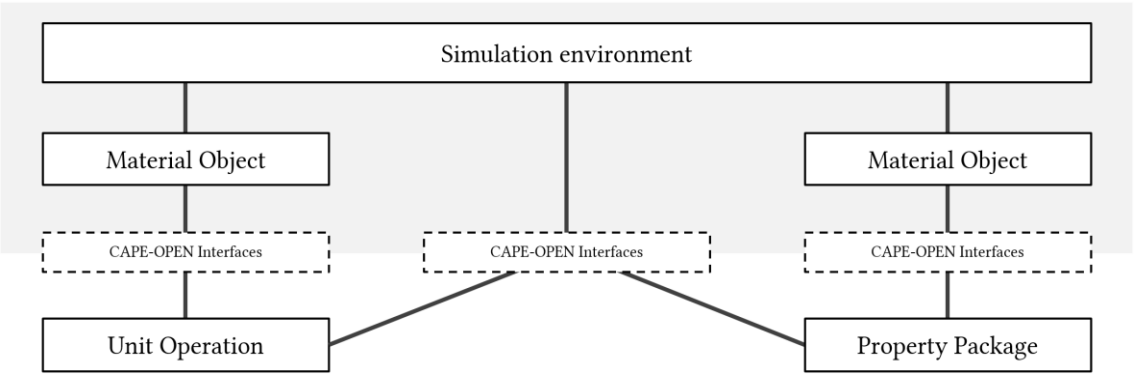


Figure 2.9 Material Object ownership (adapted from van Baten and Barrett 2009).

As mentioned above, a Material Object is connected to a material port of a unit operation. This is one of the two major applications of Material Objects. For example, a pump unit operation with a single inlet port and a single outlet port would have its inlet port connected to the Material Object that contains the feed conditions. The pump PMC obtains the conditions at the inlet before thermodynamic calculations can be performed. During the calculation procedure, the unit operation has to specify sufficient conditions at the outlet port. Once the pump PMC has set the properties on the outlet port, it instructs the Material Object connected to the outlet port to perform a thermodynamic equilibrium calculation. (van Baten and Barrett 2009)

However, a unit operation may not set properties or perform calculations on Material Objects that are connected to its inlet ports. It has to duplicate the Material Object, and perform the calculations on the duplicate. Each outlet port that is connected to the unit operation must be flashed (equilibrium calculations must be performed). Not all ports may need to be connected. For example, a mixer unit operation may have 10 inlet ports of which only a subset need to be connected. The unit operation model is usually programmed so that it checks the validity of all connections during its validation routine. (van Baten and Barrett 2009; CO-LaN 2011)

The second major use for a Material Object is to manage communication with a Property Package. The first step, in a typical order of operation of a calculation routine utilizing a Property Package, is that the simulation application sets the required inputs for the calculation

on the Material Object. The required inputs depend on the type of property calculation to be performed. For a vapour enthalpy calculation, for instance, the inputs would include pressure, temperature and composition. Next, the simulation application requests the Property Package to carry out the calculation, and at the same time tells the Property Package which Material Object carries the necessary input values. The Property Package performs the calculation based on these input values, and stores the calculation results on the Material Object. After the calculations, control is returned to the simulation application. (van Baten and Barrett 2009)

When the simulation application communicates with the unit operations and the property packages, it can do so in two ways: directly and via Material Objects, as implied in Figure 2.9. An example of direct communication with a unit operation would be to ask the unit operation to perform a calculation, or to ask for the collection of the unit operation's ports. However, all information related to streams going in or coming out of the unit operation is communicated through Material Objects, and the same applies to communications with a Property Package. Note that all interactions between the simulation application and the PMCs go via CAPE-OPEN interfaces. On the other hand, the simulation application can invoke methods on the Material Object through CAPE-OPEN interfaces, but it can access the variables and functions of the Material Object directly, too. This kind of native communication between a simulation environment and the Material Objects is usually favoured, since it is generally easier to implement, and moreover, much more efficient. (van Baten and Barrett 2009)

Depending on the application, the Material Object can be implemented in different ways. A common situation is that there is an existing simulation application, whose functionality is to be extended to include the handling of CAPE-OPEN objects. In other words, a representation of a material stream is already present in the application. The CAPE-OPEN Material Object to be created will effectively be a wrapper between the original stream object's data and methods and CAPE-OPEN. Another situation is that the Material Object is expected to be able to perform only thermodynamic calculations using a CAPE-OPEN property package. The latter type of Material Object can be rather simple, as the specifications for such an object are less extensive. (van Baten and Barrett 2009)

If the Material Object is expected to be of the type that can be passed to unit operations as well as property packages, it must implement a number of tasks, such as exposing a set of compounds and phases, having the ability to store property values, being able to delegate calculation calls to Property Packages, making a conversion of basis, keeping track of its state, and keeping track of the error status of the last method call. (van Baten and Barrett 2009)

Typically, the information about compounds present in the Material Object includes the compound ID, name, CAS registry number, normal boiling point and molecular weight. The most important of these is the compound ID, which is a string value that must be unique for each compound. The compound ID is used in the communication with a Material Object and PME to identify compounds. Nonetheless, the compound ID can have the same value as the compound name. (van Baten and Barrett 2009)

The ability of Material Objects to store the values of thermodynamic and physical properties is defined in the CAPE-OPEN interfaces *ICapeThermoMaterialObject* or *ICapeThermoMaterial*. *ICapeThermoMaterialObject* is part of CAPE-OPEN interface specifications version 1.0 and *ICapeThermoMaterial* is part of version 1.1 of the interface specifications. The newer version

(version 1.1) has its functionality divided in an improved structural manner, as shown in Figure 2.10. The property values that are stored on the Material Object represent the state of a stream in the flowsheet. As described above, the stored property values serve as inputs and outputs for the unit operations, in addition to serving as the inputs and outputs for the thermodynamic calculation performed by the Property Packages. (van Baten and Barrett 2009)

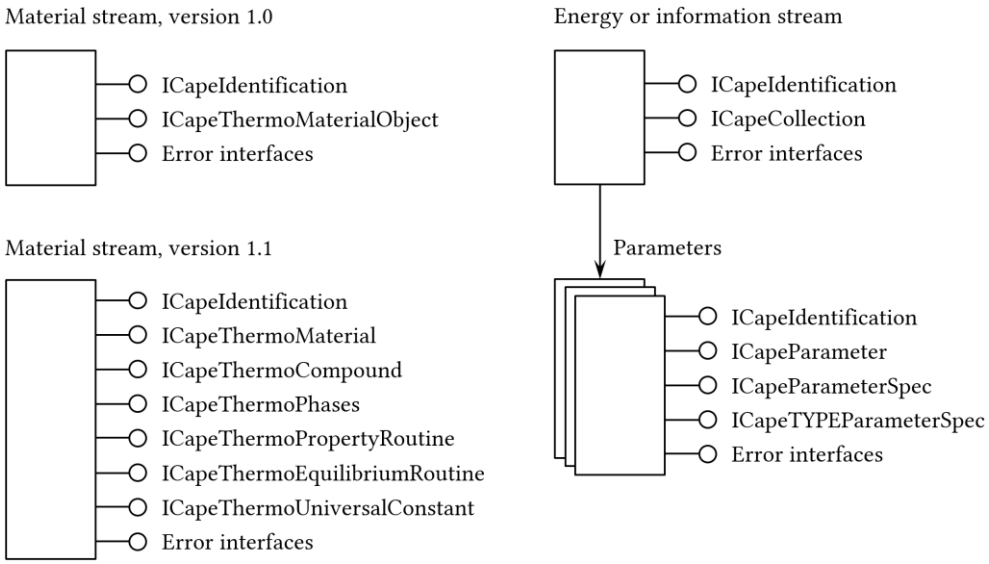


Figure 2.10 Overview of CAPE-OPEN interfaces to be implemented by objects that represent streams (adapted from van Baten 2009).

With regard to the example presented above, in which a unit operation requested a Material Object to calculate vapour enthalpy, the Material Object delegated the request to a Property Package. As a unit operation never interacts directly with a Property Package, it is not even aware that there is a Property Package used for performing thermodynamic calculations. The unit operation only makes the request to the Material Object, and if there is no Property Package, the requested calculation must be performed by the simulation application, in which case the calculation procedure details are simulator-dependent. (van Baten and Barrett 2009)

Conversion of basis is also a task of the Material Object. Conversion of basis is regarded as one of the more complicated programming tasks when implementing a Material Object. Essentially, a property either has a basis or it does not. If it has a basis, it is either molar or mass. Basis is identified with the string “mole” or “mass”, or alternatively with an empty string. For more information, see (van Baten and Barrett 2009).

The simulator needs to know at various points if a Material Object is in a state of equilibrium. For example, the simulator has to check that the Material Object representing inlet streams are ready (in equilibrium) for the unit operation to perform its calculations. Similarly, the simulator has to run a post-calculation check to see whether the unit operation has taken care of its responsibility of flashing all outlet streams. In addition, the Material Object has to keep

track of which phases are present in the material stream. To this end, version 1.1 of the CAPE-OPEN interface specifications defines a function for obtaining a list of the phases that are present, and their status. (van Baten and Barrett 2009)

As with every other CAPE-OPEN object, the Material Object is obliged to record error data when an error takes place. All errors that occur must result in a CAPE-OPEN error code, which is a number. The object that called the error producing method on the Material Object retrieves more information on the error, such as a textual description, through the CAPE-OPEN error interfaces (included in Figure 2.10). (van Baten and Barrett 2009)

To conclude this section, a general illustration of the architecture of a CAPE-OPEN compliant simulation application by Barrett and Yang (2005) is presented in Figure 2.11.

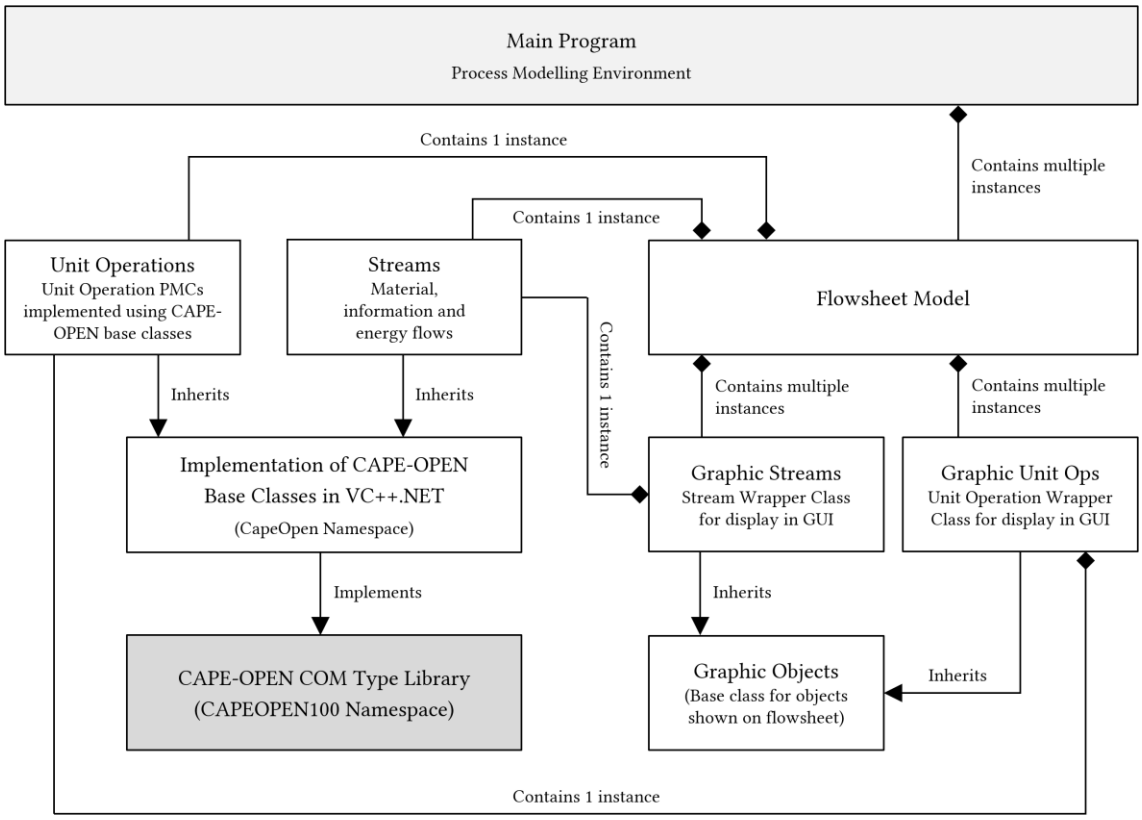


Figure 2.11 General structure of a simulator (adapted from Barrett and Yang 2005).

The application consists of two parts, PMC class libraries and a PME, which includes the graphical user interface (GUI). The component class libraries were built upon the base classes that implement the functionality specified for the appropriate CAPE-OPEN interfaces. The derived unit operation and thermodynamic modelling classes, such as the Material Object discussed above, were created from the CAPE-OPEN base classes for use within the GUI to simulate specific (metal finishing) processes. Barrett and Yang's (2005) implementation of the unit operation and material object classes is further examined in the following sections.

2.3.2 Documentation

The CAPE-OPEN interface specifications are available as a downloadable document set from www.colan.org, for free. The organization of the document set is shown in Figure 2.12.

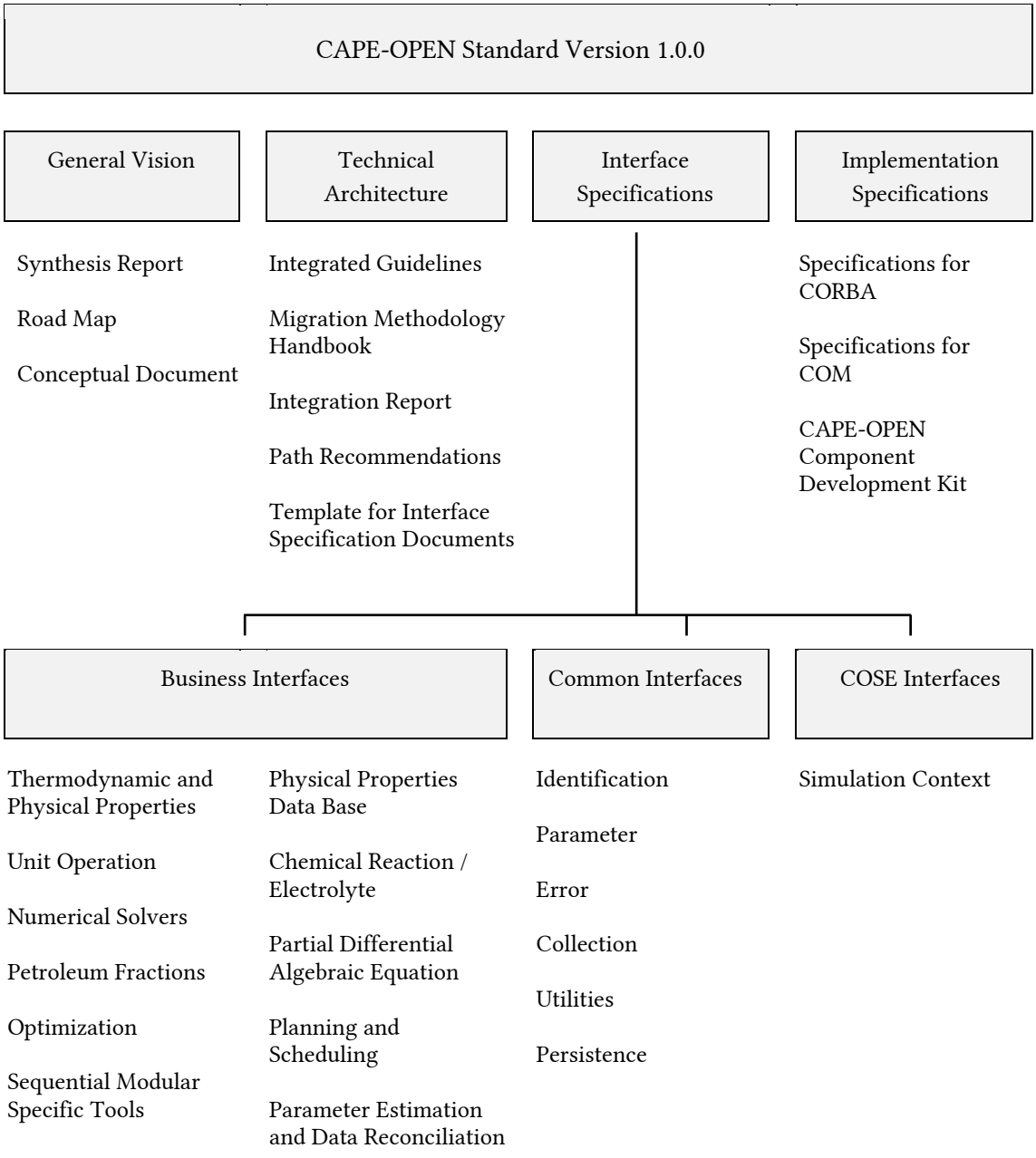


Figure 2.12 CAPE-OPEN formal documentation set (adapted from CO-LaN 2003a).

While everything under the headings General Vision, Technical Architecture and Interface Specifications is distributed as pdf documents, Implementation Specifications include middle-ware-specific binary files and software tools. The most relevant documents in this context are

the Unit Operation document, which is listed under the Business Interfaces heading, and the Specifications for COM files. In order to develop a working CAPE-OPEN compliant unit operation, one has to become familiar with virtually all the documents under the Common Interfaces heading, and the Thermodynamic and Physical Properties document, too.

In general, the CAPE-OPEN documents have a highly conceptual approach to the designing of various interfaces, some of which are presented schematically in Figure 2.13. Evidently, the idea at the planning stage was not to commit to any particular middleware model.

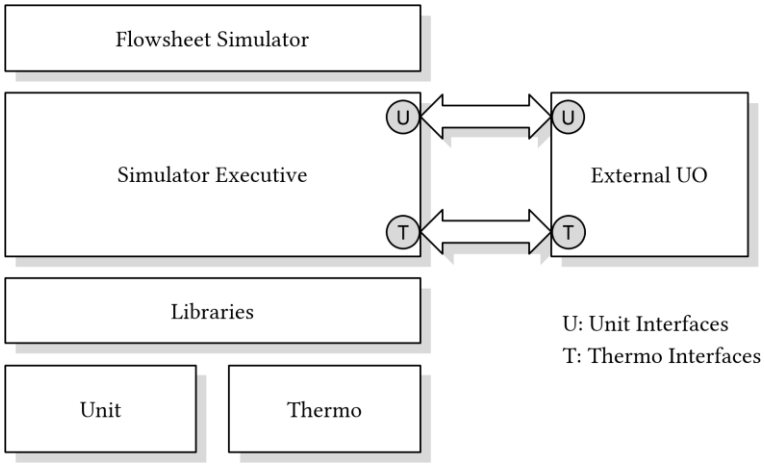


Figure 2.13 Communications through interfaces (adapted from CO-LaN 2011).

The Unit Operation document starts with a textual description of the requirements for a unit operation and then advances to use unified modelling language (UML) tools to express these requirements. The interface design is visualized with use cases, sequence diagrams, state diagrams and collaboration diagrams, and an interface diagram that is shown in Figure 2.14.

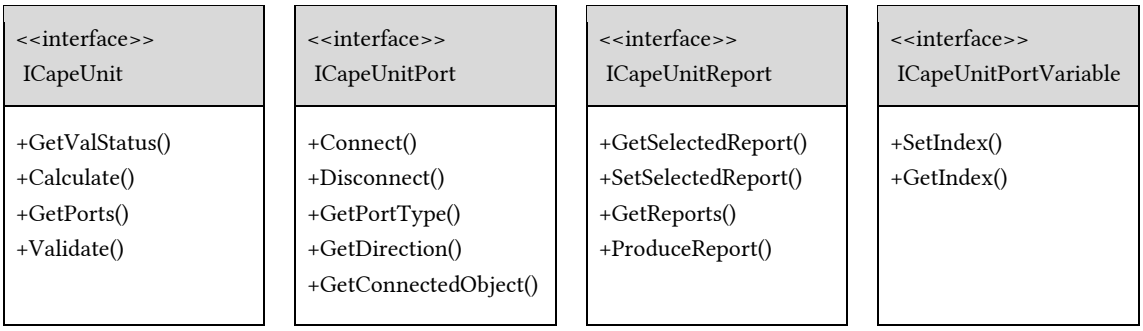


Figure 2.14 Interface diagram for a CAPE-OPEN unit operation
(adapted from CO-LaN 2011).

Every method in the interfaces shown in Figure 2.14 is described in a form similar to the one shown in Figure 2.15. The description contains the interface and method names, and the data type that the method returns, if it returns anything. There is also a brief verbal description of the method. In addition, there is information about the arguments for the method, such as their types and whether they are returned by reference ([out, return]), or are used only as input parameters ([in]). The descriptions in Figure 2.15 contain items like *ICapeCollection* and *ECapeUnknown*, which are described in the Collections and Error documents, respectively.

Interface Name	ICapeUnit
Method Name	GetPorts
Returns	CapeInterface
Description	
Returns an interface to a collection containing the Ports (e.g. ICapeCollection) of the Unit Operation.	
Arguments	
Name	[out, return] portsInterface
Type	CapeInterface
Description	A reference to the interface on the collection containing the ports.
Errors	
ECapeUnknown	

Figure 2.15 Description for the *ICapeUnit.GetPorts* method (adapted from CO-LaN 2006).

These descriptions are the bases used when the actual interface definition files were created. The interfaces were defined in Microsoft's interface definition language (MIDL) and in the CORBA interface definition language (CIDL) for use with COM and CORBA middleware, respectively. These files are available for download from www.colan.org, as stated earlier.

A MIDL file has two parts: an interface header and interface body. One of the attributes in the header is the universally unique identifier (UUID), which distinguishes it from other interfaces. The interface definition for the *ICapeUnit* interface that was shown in Figure 2.14 and partially described in Figure 2.15 is presented overleaf in Figure 2.16.

The interface header and the interface body contain COM specific details which are beyond the scope of this work. However, to be useful in a development project, the MIDL file has to

be compiled into a type library (a .tlb file) by using a MIDL compiler. In fact, CO-LaN provides an already compiled type library, CAPE-OPENv1-0_0.tlb, for download on their website. Before being able to use the COM interfaces in a .NET project, the COM specific type library data has to be converted to suitable .NET metadata. This can be done with the type library importer tool, tlbimp.exe, which produces an interop assembly, CAPEOPEN100.dll, which enables .NET Framework applications to use all the interfaces, enumerations and types contained in the type library.

```
//        This interface provides the basic functionality for a Unit
//        Operation component
[
          object,
          uuid(ICapeUnit_IID),
          dual,
          helpstring("ICapeUnit Interface"),
          pointer_default(unique)
]
interface ICapeUnit : IDispatch
{
          //  Get the collection of unit operation ports
          //
          //  CAPE-OPEN exceptions:
          //  ECapeUnknown,  ECapeFailedInitialisation,  ECapeBadInvOrder
          [propget, id(1), helpstring("Gets the whole list of ports")]
          HRESULT ports([out, retval] CapeInterface* ports);

          //  Gets the flag to indicate unit's validation status
          //  · notValidated(0),invalid(1) or valid(2)
          //
          //  CAPE-OPEN exceptions
          //  ECapeUnknown, ECapeInvalidArgument
          [propget, id(2), helpstring("Get the unit's validation status")]
          HRESULT ValStatus([out, retval] CapeValidationStatus *valStatus);

          //  Executes the necessary calculations involved in the unit
          //  operation model
          //
          //  CAPE-OPEN exceptions raised:
          //  ECapeUnknown,  ECapeBadInvOrder,  ECapeOutOfResources,  ECapeTimeOut,
          //  ECapeSolvingError,  ECapeLicenceError
          [id(3), helpstring("Performs unit calculations")]
          HRESULT Calculate();

          //  Validate that the parameters and ports are all valid
          //
          //  CAPE-OPEN exceptions:
          //  ECapeUnknown,  ECapeBadCOParameter,  ECapeBadInvOrder
          [id(4), helpstring("Validate the Unit")]
          HRESULT Validate([ACTUALLYout] CapeString* message, [out, retval] CapeBoolean* isValid);
};
```

Figure 2.16 COM MIDL code for the *ICapeUnit* interface.

Many COM techniques are used in conjunction with the CAPE-OPEN interfaces, even if the application development of a simulation environment was done on the .NET framework. This requires the developer to obtain a certain familiarity with COM. Amendments to the CAPE-OPEN document set have been introduced (CO-LaN 2006) to provide information concerning COM and .NET interoperability. Some key aspects regarding COM are also briefly presented later in the text. For example, Section 2.3.3 contains information about component registration using the Windows registry, and Section 3.2.3 discusses interoperability and the differences between COM and .NET error handling mechanisms.

2.3.3 Requirements for unit operations

This section introduces some of the basic requirements for developing a standard compliant unit operation, which come with a number of associated objects that are defined in the CAPE-OPEN standards. The associated objects include the unit operation ports and parameters as shown in Figure 2.17. For illustrative purposes, Figure 2.17 also includes a screenshot of the DWSIM implementation of a properties editing panel for a CAPE-OPEN unit operation.

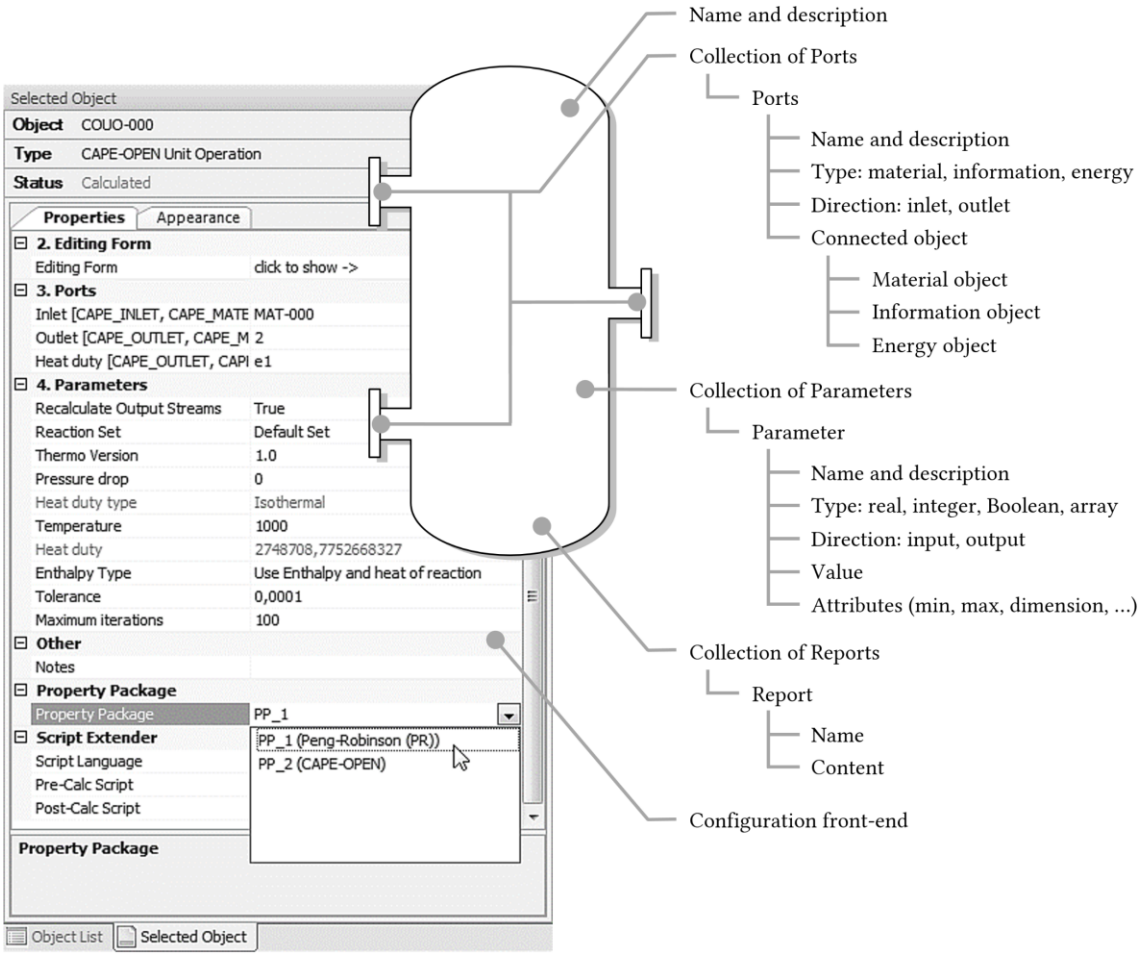


Figure 2.17 CAPE-OPEN unit operation and associated objects (adapted from van Baten and Pons 2014; Medeiros et al. 2014a).

The unit operations in CAPE-OPEN are treated with a sequential modular approach. The inputs and outputs of the unit operation are identified, and it is the responsibility of the unit operation to provide values for all its outputs. Also, ports and parameters are assigned a direction. While executing calculations, a unit operation must specify values for the output parameters, and for all the values related to the outlet streams. Material streams are specified by composition and flow rate. Two additional values specify the phase equilibrium, either temperature and pressure, or pressure and enthalpy. Since a unit operation specifies the phase

equilibrium at the material outlet streams, the process-modelling environment does not need to know which values the unit operation produced, as all values in the outlet streams are thermodynamically consistent. Similarly, during calculation, a unit operation can safely make an assumption that the feed values are in phase equilibrium. (van Baten and Pons 2014)

With regard to the interfaces that a unit operation developer is expected to implement, see Fig. 2.18. All CAPE-OPEN objects are required to implement the *ICapeIdentification* interface, which contains a name and a description for a unit operation, and the CAPE-OPEN error interfaces. In addition, other interfaces that are directly affiliated with a unit operation include the *ICapeUnit* and *ICapeUnitReport* interfaces, as presented in Section 2.3.2. In addition, it is advised to implement the *ICapeUtilities* interfaces, which are defined in the CAPE-OPEN Common interfaces document set, as shown in Figure 2.12. (van Baten and Pons 2014)

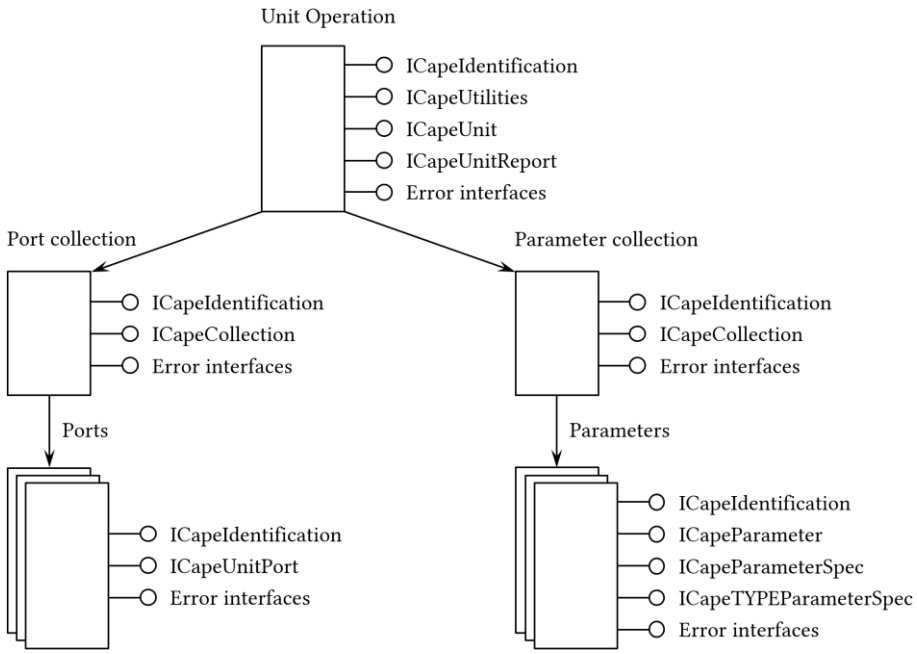


Figure 2.18 Overview of object hierarchy and CAPE-OPEN interfaces to implement for a unit operation (adapted from van Baten 2009).

As the streams present in a flowsheet are connected to a unit operation via ports, it has to have a pertinent collection of ports. Each of these ports has a direction and a type. Definitions exist for material, energy and information ports, but currently only material ports are widely supported. (van Baten and Pons 2014)

Other types of information can be exchanged with the flowsheet via the use of parameters. Parameters have a type, direction and value. Depending on the type of parameter that is used, relevant attributes exist to describe the value of the parameter. For example, for a real parameter, the attributes include a minimum and a maximum value, a default value, and the SI-based unit of measure of the value. (van Baten and Pons 2014)

Tables 2.2–2.3 present the requirements for the development of CAPE-OPEN compliant unit operations. Similar tables for thermodynamic servers are presented in (van Baten et al. 2016).

Table 2.2. Unit operation v1.0 compliancy requirements (van Baten et al. 2016)

Item	Requirement
1	The package (unit operation class) exposes a unit operation interface exposes *ICapeUnit*, from Unit Operations specification (CO-LaN 2011), registration of COM objects and implemented categories as in Methods & Tools Integrated guidelines (CO-LaN 2003a)
2	The unit operation can be validated proper implementation of *ICapeUnit.GetValStatus* and *ICapeUnit.Validate*
3	The unit operation can run proper implementation of *ICapeUnit.Calculate*
4	The unit operation properly exposes ports proper implementation of *ICapeUnit.GetPorts*, and returned collection according to Collections specification (CO-LaN 2003b), each port object implementing *ICapeUnitPort* and identification in accordance with Identification Common Interface (CO-LaN 2003c)
5	The ports can be connected and disconnected proper operation of *ICapeUnitPort.Connect* and *ICapeUnitPort.Disconnect*
6	Port's connected objects are returned when asked proper operation of *ICapeUnitPort.GetConnectedObject*
7	Type and direction of ports can be obtained proper implementation of *ICapeUnitPort.GetDirection* and *ICapeUnitPort.GetPortType*
8	The unit operation's user interface is implemented proper implementation of *ICapeUtilities.Edit*
9	If present, proper implementation of reports proper implementation of *ICapeUnitReport.GetSelectedReport*, *ICapeUnitReport.SetSelectedReport* and *ICapeUnitResport.GetReports*

The class that will represent the unit operation is required to have an implementation for the methods listed in items 1–3 in Table 2.2, and it has to be registered as a COM object, see item

15 in Table 2.3. Additional classes have to be created to implement the port and parameter collections. Also, one or more classes have to be written to represent the ports, as items 4–7 in Table 2.2 imply. A port class has to implement the *ICapeUnitPort* interface. Concerning items 8 and 9, the implementation of *ICapeUtilities* and *ICapeUnitReport* interfaces is optional.

Table 2.3. Unit operation v1.0 compliancy requirements (van Baten et al. 2016)

Item	Requirement
10	The unit operation can be loaded into COFE (or another compliant simulator)
11	The unit parameters (if present) are exposed correctly by COFE if parameters are present, implementation of Utilities Interface (CO-LaN 2003d) *ICapeUtilities*, proper operation of *ICapeUtilities.GetParameters*, proper operation of the returned collection according to Collections specification (CO-LaN 2003b), parameter implementation according to Parameter specification (CO-LaN 2003e)
12	An equilibrium calculation is performed on each of the connected outlet ports during calculation
13	Persistence is implemented so that the unit operation is properly loaded or stored (unless no persistence is required for proper operation) *IPersistStream* or *IPersistStreamInit* implementation, in accordance with Persistence specification (CO-LaN 2003f)
14	Component identification is implemented in accordance with Identification Common Interface (CO-LaN 2003c)
15	Version information is exposed registry entries as described in Method & Tools Integrated guidelines (CO-LaN 2003a)
16	Errors are presented as CAPE-OPEN HRESULT error codes and appropriate error interface is implemented in accordance with Error Common Interface (CO-LaN 2003g)
17	No errors occur that prevent proper operation of the unit operation

If the unit operation is to have any parameters, item 11 in Table 2.3, one or more classes that expose the *ICapeParameter* and relevant parameter type interfaces, such as *ICapeRealParameterSpec or ICapeIntegerParameterSpec* have to be implemented. If a unit operation is expected to have the ability to save and load its state, it is required to implement either the *IPersistStream* or the *IPersistStreamInit* interface. (CO-LaN 2006; van Baten et al. 2016).

As remarked in item 1 in Table 2.2 and in item 15 in Table 2.3, a unit operation class needs to be registered as a COM object. The registration procedure involves an addition of attributes that are inserted above the class signatures, as shown in Fig. 2.19. In the example in Fig. 2.19, the first attributes tell us that the class can be serialized onto memory; the second attribute makes the class COM visible, and the last attribute assigns the class a globally unique identifier (GUID) that makes the class identifiable across the system in which it has been registered.

```
[
   Serializable, ComVisibleAttribute(true), GuidAttribute("6870DFAF-746E-4DFB-A26F-1261DE7B17EE")
]
public ref class CParameterCollection : public BindingList<ICapeParameter^>,
   public ICapeIdentification, ICapeCollection, ICapeUnitCollection
{
  // …
}
```

Figure 2.19 Attributes for a parameter collection (adapted from CO-LaN 2006).

Figure 2.20 shows an implementation for a class registration written in Visual Basic. The .NET Framework include methods, such as *CreateSubKey*, for creating Windows registry keys and setting values in them. The code snippet in Figure 2.20 is from the source code for DWSIM.

```
<System.Runtime.InteropServices.ComRegisterFunction()> _
Private Shared Sub RegisterFunction(ByVal t As Type)

   Dim keyname As String = String.Concat("CLSID\\{", t.GUID.ToString, "}\\Implemented Categories")
   Dim key As Microsoft.Win32.RegistryKey = Microsoft.Win32.Registry.ClassesRoot.OpenSubKey(keyname, True)
   If key Is Nothing Then
      key = Microsoft.Win32.Registry.ClassesRoot.CreateSubKey(keyname)
   End If
   key.CreateSubKey("{678C09A5-7D66-11D2-A67D-00105A42887F}") ' CAPE-OPEN Unit Operation
   key.CreateSubKey("{678C09A1-7D66-11D2-A67D-00105A42887F}") ' CAPE-OPEN Object
   keyname = String.Concat("CLSID\\{", t.GUID.ToString, "}\\CapeDescription")
   key = Microsoft.Win32.Registry.ClassesRoot.CreateSubKey(keyname)
   key.SetValue("Name", "IronPython/Lua Scripting Unit Operation")
      ' ...
   key.Close()

End Sub

Private Sub RegistrarTiposCOMToolStripMenuItem_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles RegistrarTiposCOMToolStripMenuItem.Click

   Dim windir As String = Environment.GetEnvironmentVariable("SystemRoot")
   Process.Start(windir & "\Microsoft.NET\Framework\v4.0.30319\RegAsm.exe", "/codebase /silent " & _
   Chr(34) & My.Application.Info.DirectoryPath & " \DWSIM.exe" & Chr(34))

End Sub
```

Figure 2.20 Creating registry keys (top), and calling regasm.exe (bottom).

The second code sample in Figure 2.20 uses the .NET assembly registration tool regasm.exe to add the necessary entries to the registry. Once a .NET class like the one in Figure 2.20 is registered, it can be used by applications as if it were a COM class. (Microsoft 2016a)

Fig. 2.21 shows the information that the Windows registry holds for a CAPE-OPEN compliant mixer unit operation, after the registration procedure. The class identification GUID (CLSID), at the top of Figure 2.21 is the same as the GUID mentioned earlier, albeit marked differently.

```
CLSID: {5E216BDD-C82F-45FE-830C-61FDDD2D9BC7}

ProgID: COCO_COUS.Mixer

Name: Mixer
CapeVersion: 1.0
ComponentVersion: 3.0.0.10
VendorURL: http://www.cocosimulator.org/
Description: Mixer - unit operation to adiabatically mix 2 or more inlet streams into a single outlet
About: CAPE-OPEN 1.0 unit operation - Copyright 2015 www.cocosimulator.org

Implemented categories:
Supports Thermodynamics 1.0 CATID {0D562DC8-EA8E-4210-AB39-B66513C0CD09}
Consumes Thermo CATID {4150C28A-EE06-403f-A871-87AFEC38A249}
Supports Thermodynamics 1.1 CATID {4667023A-5A8E-4CCA-AB6D-9D78C5112FED}
CAPE-OPEN Component {678c09a1-7d66-11d2-a67d-00105a42887f}
CAPE-OPEN Unit Operation {678c09a5-7d66-11d2-a67d-00105a42887f}

InprocServer32: C:\Program Files\COCO\COCOCOUS.dll
ThreadingModel: Apartment

TypeLib: {2416578B-1867-4CD9-B71F-4AE0C05FB06C}

Registry location: HKEY_LOCAL_MACHINE\Software\Classes\CLSID\{5E216BDD-C82F-45FE-830C-61FDDD2D9BC7}
```

Figure 2.21 Windows registry information about a mixer unit operation (obtained with CORK, see Section 2.4.1).

As shown in Figure 2.21, the COM registration information for a typical unit operation also includes the assignment of the CAPE-OPEN Unit Operation category IDs, some of which are listed in Table 2.4. If relevant, additional information regarding the support for different versions of thermodynamics interfaces is included as well. (van Baten et al. 2016)

Table 2.4. GUIDs for CAPE-OPEN component categories (CO-LaN 2006)

Category	Globally Unique Identifier
CAPE-OPEN Component	{678C09A1-7D66-11D2-A76D-00105A42887F}
.	.
.	.
.	.
CAPE-OPEN Unit Operation	{678C09A5-7D66-11D2-A76D-00105A42887F}
CAPE-OPEN Thermo Equilibrium Server	{678C09A6-7D66-11D2-A76D-00105A42887F}

For more information on registering object for use with CAPE-OPEN compliant simulation environments, see for example (CO-LaN 2006), (CO-LaN 2011) and (van Baten et al. 2016).

2.3.4 Creation of unit operation classes

This section illustrates the development of compliant unit operation classes with an example implementation that exposes the required CAPE-OPEN interfaces. Since the developed class is made to work in the .NET framework, but is also expected to work within COM, additional information on .NET and COM interoperability is provided. The section concludes with an introduction to the methods that have been created to help develop compliant unit operations.

Object-oriented implementation

The discussion here is based on the class implementations made by Barrett and Yang (2005). Some of the key CAPE-OPEN interfaces involved and the structure of their class hierarchy for a unit operation model are shown in Figure 2.22.

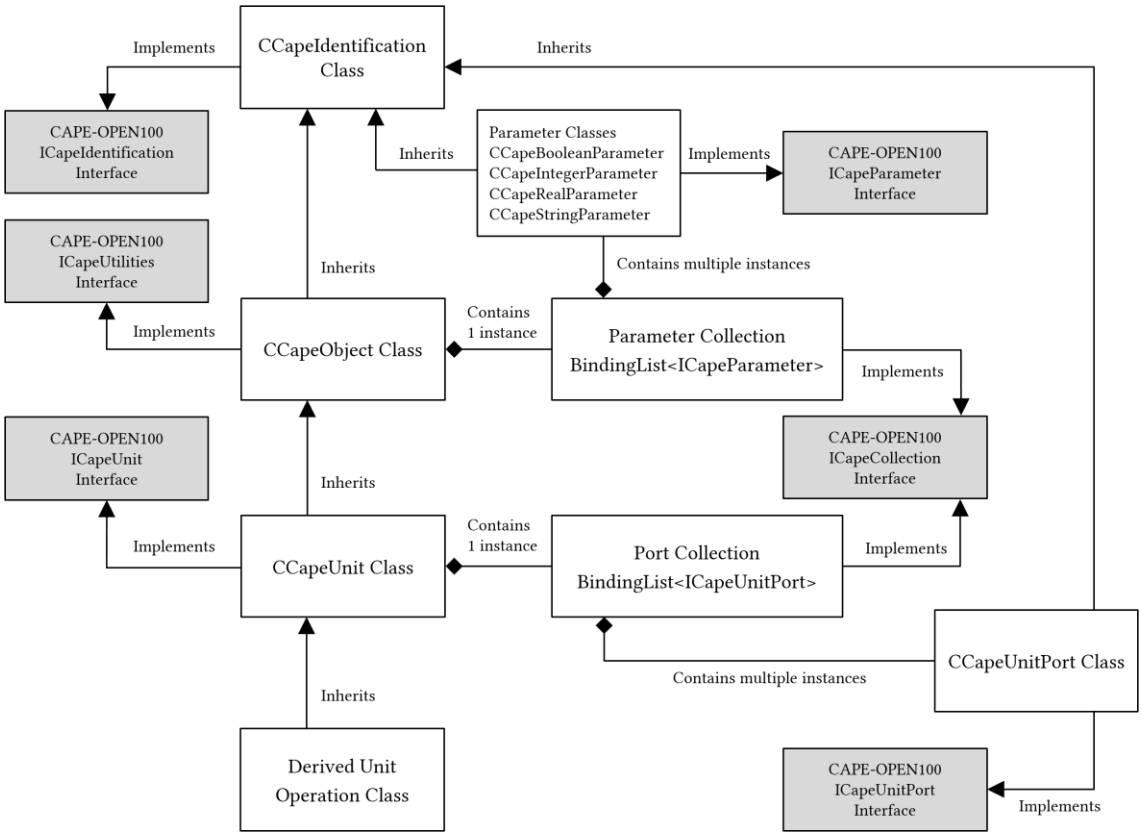


Figure 2.22 Model of unit operation classes that derive from the *CCapeUnit* class (adapted from Barrett and Yang 2005).

When Barrett and Yang started to create their simulation application, they first imported the CAPE-OPEN Version 1.0 COM type library into a .NET primary interop assembly (PIA). The type library is available from www.co-lan.org, as described in Section 2.3.2. This process created a CAPEOPEN100 namespace that contains the CAPE-OPEN interface definitions, some

of which are shown in Figure 2.22 as dark grey rectangles, and provided a built-in marshalling of data from the CAPE-OPEN COM objects to the .NET framework using the data types indicated in Table 2.11 on page 63. (Barrett and Yang 2005)

Barrett and Yang created classes that implemented the CAPE-OPEN interfaces. The *CCapeIdentification* class, which implemented the *ICapeIdentification* interface (CO-LaN 2003c), exposed the *ComponentName* and *ComponentDescription* properties in *ICapeIdentification*. Consequently, *CCapeIdentification* was inherited by all the modelling component classes that were created, to provide a component name and description for the class. (Barrett and Yang 2005)

The required parameter classes were created as wrappers around the appropriate .NET type. For instance, the *CCapeRealParameter* class was used to contain the value of the parameter as a double-precision number. The *CCapeRealParameter* class implemented the *ICapeParameter*, *ICapeParameterSpec* and *ICapeRealParameterSpec* interfaces in order to comply with the CAPE-OPEN specifications (CO-LaN 2003e). (Barrett and Yang 2005)

Also, type-safe parameter and port collection classes were created in order to have a container for parameters and ports. The collection classes implemented the CAPE-OPEN *ICapeCollection* interface (CO-LaN 2003b), and derived from the .NET framework *BindingList* class template. The BindingList class template is a generic collection that supports data binding, and provides implementation of the .NET framework *IList*, *ICollection* and *IEnumerable* interfaces. The enumerator facilitates iteration through all items of the collection. Since the collections were type-safe, it was assured that the port and parameter collections contained only ports and parameters, respectively. The collections contained items that were either *ICapeUnitPorts* or *ICapeParameters*. The use of the CAPE-OPEN *ICapeCollection* interface made the collections accessible from COM. (Barrett and Yang 2005)

Figure 2.11 illustrates the general structure of the simulation application developed in (Barrett and Yang 2005). The base classes shown in the schematic diagram in Figure 2.11 include the *CCapeObject* class, which serves as a base class for all process modelling components such as unit operation, material object, information object and energy object classes. As shown in Figure 2.22, the *CCapeObject* class inherits the *CCapeIdentification* class and implements the *ICapeUtilities* (CO-LaN 2003d) interface. Also, *CCapeObject* includes a member variable named *m_parameters*, which is the above-described collection of parameters. (Barrett and Yang 2005)

As shown in Figure 2.22, the base class for specialised unit operations – the *CCapeUnit* class – inherits the *CCapeObject* class and implements the *ICapeUnit* (CO-LaN 2011) interface. It also contains a collection of *CCapeUnitPort* objects. The *CCapeUnitPort* class implements the *ICapeUnitPort* interface, and contains a collection of port variables, which are themselves parameters. A derived unit operation class can then inherit the *CCapeUnit* that includes the necessary port and parameter collections, and the implementations for all the required CAPE-OPEN interfaces. (Barrett and Yang 2005)

Figure 2.23 contains the Visual Basic code listing that implements a mixer unit operation. The constructor of the class creates and adds ports and parameters into the appropriate collections. A single parameter of type *CCapeRealParameter*, called TestParam, and a value of 100 are added to the unit operation. The parameter is then placed in the parameter collection by casting the parameter collection to the *IList* data type and calling the *IList.Add* method. The

ports are created by calling the constructor for the *CCapeUnitPort* class with the name, description, type and direction of the port passed in as parameters. Port type and port direction are enumerated variables defined in the CAPE-OPEN unit operations specification, and can have the values CAPE_INLET, CAPE_OUTLET or CAPE_ INLET_OUTLET for the port direction, and the values CAPE_MATERIAL, CAPE_ENERGY, CAPE_INFORMATION or CAPE_ANY for the port type. Once created, the port is added to the ports collection through the *IList.Add* method. (Barrett and Yang 2005)

```
Imports CapeOpen
Imports CAPEOPEN100
Imports MetalFinishing
Imports System

<Serializable()> Public Class VBUnitObject
    Inherits CCapeUnit
    Public Sub New()
        Me.m_ComponentName = "Visual Basic Mixer Unit Operation"
        Me.m_ComponentDescription = "The unit operation demonstrates use of a unit operation written in VB"
        Dim param As CCapeParameter = New CCapeParameter("TestParam", 100)
        Dim params As IList = Me.parameters
        params.Add(param)
        Dim ports As IList = Me.ports
        Dim objectType As CAqueousMaterialObject
        Dim port As CCapeUnitPort
        port = New CCapeUnitPort("Inlet Port 1", "Inlet Port 1 for VB Mixer", CAPE_MATERIAL, CAPE_INLET)
        port.ConnectedObjectType = objectType.GetType
        ports.Add(port)
        port = New CCapeUnitPort("Inlet Port 2", "Inlet Port 2 for VB Mixer", CAPE_MATERIAL, CAPE_INLET)
        port.ConnectedObjectType = objectType.GetType
        ports.Add(port)
        port = New CCapeUnitPort("Outlet Port", "Outlet Port for VB Mixer", CAPE_MATERIAL, CAPE_INLET)
        port.ConnectedObjectType = objectType.GetType
        ports.Add(port)
    End Sub
    Public Overrides Sub Calculate()
        Dim ports As ICapeCollection
        Dim port As ICapeUnitPort = ports.Item(0)
        Dim in1 As CAqueousMaterialObject = port.connectedObject
        If (in1 Is Nothing) Then
            Throw New CapeUnknownException("Material object connected to Inlet Port 1 is not valid.")
        End If
        port = ports.Item(1)
        Dim in2 As CAqueousMaterialObject = port.connectedObject
        If (in2 Is Nothing) Then
            Throw New CapeUnknownException("Material object connected to Inlet Port 2 is not valid.")
        End If
        port = ports.Item(2)
        Dim outObject As CAqueousMaterialObject = port.connectedObject
        If (outObject Is Nothing) Then
            Throw New CapeUnknownException("Material object connected to the Outlet Port is not valid.")
        End If
        outObject.CombineStreams(in1, in2)
    End Sub

End Class
```

Figure 2.23 Visual Basic code implementing a mixer unit operation (adapted from Barrett and Yang 2005).

After the unit operation has been implemented, the GUI is used to connect different material objects to the correct ports on the mixer unit operation. The *CAqueousMaterialObject* that was developed for the project in (Barrett and Yang 2005) is the connected object in this case. As shown in Figure 2.23, the unit operation's *Calculate* method obtains pointers to the two inlet streams and the single outlet stream using the *ICapeUnitPort.connectedObject* property. Since

the unit operations constructor placed the ports into the collection in a known order, the inlet and outlet ports are obtained using specified indices. In addition, because the type of the material object is known to be *CAqueousMaterialObject*, the material object is directly cast to that type. The casting operation examines if the object is of the desired type, and returns a valid pointer only if the material objects are of the requested type. Thus, if the material objects attached to any of the ports is not of type *CAqueousMaterialObject*, the pointer will be null (*Nothing* in Visual Basic) and the calculation routine will throw a *CapeUnknownException*, which contains a message indicating that the object connected to the port is not a material object of the right type. The *CombineStreams* method in the *CAqueousMaterialObject* class is an extension of the material object beyond the minimal functionality required in the CAPE-OPEN specification. Additionally, the same implementation of the mixing algorithm in the *CombineStreams* method can be used whenever similar combining of streams is required. The mixing algorithm first determines the concentration of components and flow rates of the inlet streams and then calculates the resulting concentrations and flows for the outlet stream using the material and energy balance. (Barrett and Yang 2005)

All of the base classes are marked with the *Serializable* attribute. Serialization is the process of converting an object into a stream of bytes in order to save the object to memory, database or a file. The reverse process is called deserialization. All the derived classes, such as the *VBUnitObject* in Figure 2.23, must also be marked with the same attribute in order to be serialized. Barrett and Yang (2005) had their application serialize the flow sheet as an extensible mark-up language (XML) document by the appropriate .NET framework's XML serializer class. In fact, the use of the *Serializable* attribute also enables the .NET framework to serialize objects to remote computers. (Barrett and Yang 2005)

The primary goal of the unit operation *CCapeUnit* object base class was to implement as much of the generic functionality of the CAPE-OPEN interface specification as was practical, essentially encapsulating this functionality. Functions whose action may be different in different unit operations were declared *virtual* (*overridable* in Visual Basic), which means that the derived class implementations replace the base class implementation. The *ICapeUtilities.Edit()* method serves as an example of a method defined as *virtual* in the base class, but that may be implemented in the derived class. As a default, the *Edit()* function throws a *CapeNoImplException* exception, which is caught by the flowsheet application as a signal that the *Edit*() function has not been implemented and that the application's built-in unit operation editor, which provides a list of parameters and ports, should be used instead. Alternatively, if the *Edit()* function has been overridden and does not return a failure condition (throw an exception), the GUI assumes that the object was successfully edited. (Barrett and Yang 2005)

Under .NET, an application exception class called *System.ApplicationException* is available that can be used to pass information such as a message and the source of the exception. The CAPE-OPEN exception definitions all derive from the *ECapeRoot* interface (CO-LaN 2003g). Barrett and Yang (2005) implemented the CAPE-OPEN exception classes so that all exception classes derive from the *CapeRootException* class derived in turn from the *System.ApplicationException* class. The *CapeRootException* class exposes the *ECapeRoot* interface. In this way, all exceptions raised by the classes that represent process modelling components can be caught either as a *CapeRootException* or as a *System.ApplicationException* in addition to being caught as the derived exception type. (Barrett and Yang 2005)

Methods to help development

According to the feedback CO-LaN has received, CAPE-OPEN is considered a technology that is difficult to master quickly (Pons 2005). Consequently, several tools have been devised to help with the development of CAPE-OPEN compliant unit operations. Some of these tools are listed in Table 2.5, although the list is not exhaustive.

Table 2.5. Tools to aid CAPE-OPEN component development

Tool	Remark
CAPE-OPEN Unit Wizard	Developed by AspenTech. Last updated in 2004, requirements include VB6 tools. (AspenTech 2016)
CAPE-OPEN/COCO compliancy testing	Interoperability tests performed by the COCO development team at the request of the software vendor. (van Baten et al. 2016)
Consultancy scheme	Free consultancy for software developers creating CAPE-OPEN components, provided by CO-LaN. (Pons 2005). The current availability of the service (in February 2016) is unknown.
CO Tester Suite	A tool for checking a software component's conformity with CAPE-OPEN standards. Last updated in 2003. (CO-LaN 2014)
Framework (a set of classes)	A framework that contains a set of C++ classes. The intended use is prototyping and education. (Testard and Belaud 2005).
Object Pascal CAPE-OPEN Wizard	Developed by C. Crause. Last updated in 2013. Code written in Object Pascal, requirements include Delphi IDE. (Crause 2013)
Software Factory	An attempt to automatically generate compliant code, involving the use of several tools and creation of meta models. (Lajmi et al. 2009)
Rapid prototyping	Prototyping of unit operations models by using generic modelling tools MATLAB, Scilab and Excel via CAPE-OPEN. (van Baten 2009)
Unit Wizard	Also known as CAPE-OPEN Studio .NET. Creates code stubs that can be completed by the developer in C#. Also helps with COM registration. (Fermeglia and Parenzan 2007)

The last item in Table 2.5, Unit Wizard, targets .NET Framework version 2.0, but can be made to target later versions as well. It helps with COM registration, although there is scarce information as to the details of the registry entries made. The topic of development aids is revisited in Section 2.3.6, which includes more information on rapid prototyping methods.

2.3.5 Additional information

Closely connected to the implementation of standard compliant unit operations is the creation of material object and property package classes. This section provides a short introduction to the CAPE-OPEN Material Object via an example implementation, as presented in Barrett and Yang (2005). In addition, supplementary information on Property Packages is provided.

Implementation of Material Objects

Material Objects provide information about material flows within the flowsheet. Barrett and Yang created an implementation of a thermodynamic model that consisted of three parts. The centrepiece was a Material Object that was connected to a Property Package and a Reaction Package. Each of these parts was based upon its respective CAPE-OPEN interface standards. The basic structure of the material system is shown in Figure 2.27.

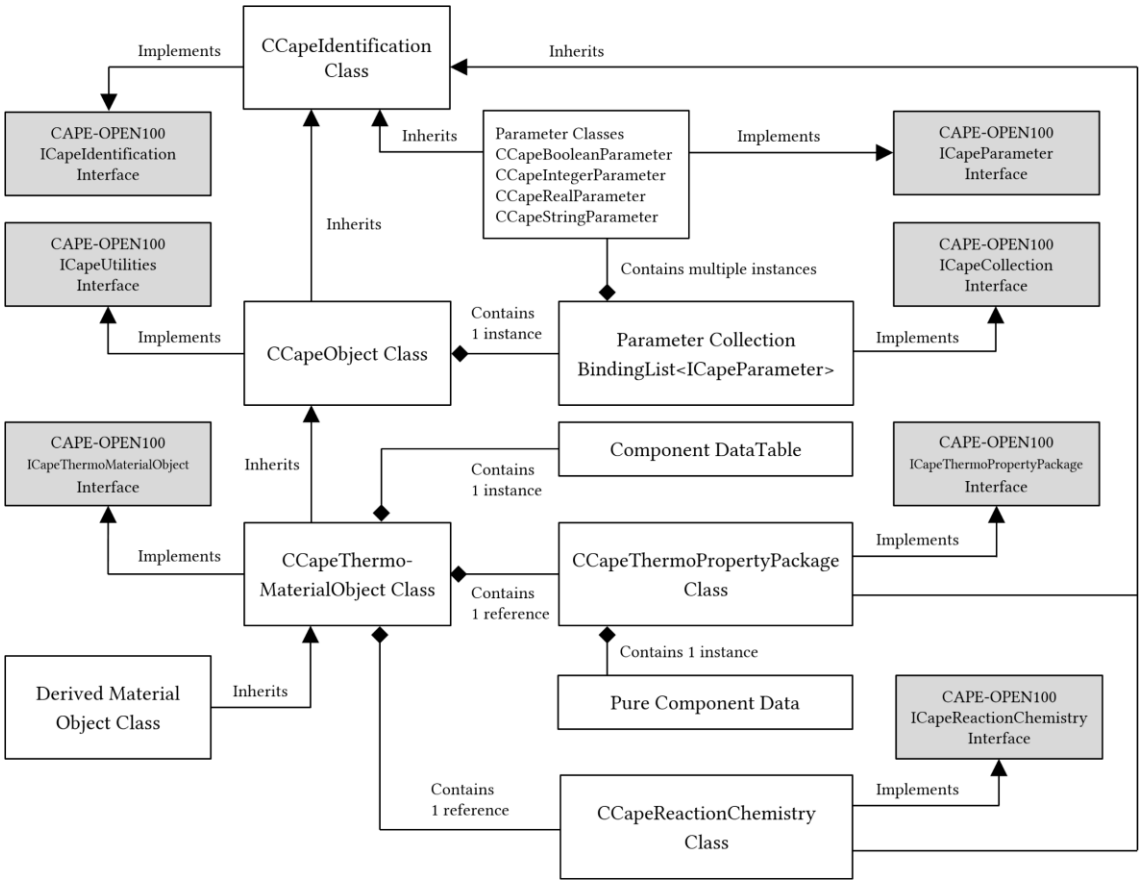


Figure 2.27 Model of material object classes that derive from the *CCapeThermoMaterialObject* class (adapted from Barrett and Yang 2005)

As shown in Figure 2.27, the *CCapeThermoMaterial* class is based on the *CCapeObject* class, and can be inherited on demand to provide conceptual models of the materials being

simulated. The *CCapeThermoPropertyPackage* and the *CCapeReactionChemistry* classes derive directly from the *CCapeIdentification* class. At the centre of the *CCapeThermoMaterialObject* and *CCapeThermoPropertyPackage* classes is a data table (.NET framework *DataTable*). The data table contains appropriate general, physical and chemical properties of the chemical species modelled. (Barrett and Yang 2005)

The *CCapeThermoMaterialObject* class provides methods for obtaining the constant properties of each chemical species from the Property Package, and for having the Property Package calculate the values of chemical properties that depend on temperature, pressure and composition. The *CCapeThermoMaterialObject* class contains a parameter collection that stores the pressure, temperature and flow rate of the Material Object. In this implementation, the parameter collection is built around a .NET framework's *BindingList,* which supports data binding to user interface controls, for example, and exposes these overall properties as properties of the object. The data table in the *CCapeThermoMaterialObject* class initially contains only the IUPAC name and concentration of each of the chemical species in the *CCapeThermoMaterialObject*. By using a chemical's IUPAC name, the *CCapeThermoMaterialObject* class can obtain the values of constant properties for the chemical directly from the Property Package, or it could request that the Property Package calculate values of non-constant properties based on the temperature, pressure and composition of the *CCapeThermoMaterialObject*. Eventually, the values of these properties can be added to the Material Object's data table for later use.

Because the *CCapeThermoPropertyPackage* class implements the *ICapeThermoPropertyPackage* interface, it has to provide functions to obtain the values of the chemical species' constant properties, and methods to calculate the equilibrium conditions and chemical properties based on the conditions of the material object that is passed to it as a parameter. The data table in the *CCapeThermoPropertyPackage* contains the various constant properties of the chemical species used in the flowsheet. In fact, the constant properties in the data table include a subset of the constant properties of the chemical listed in the CAPE-OPEN interface specification. Furthermore, Barrett and Yang implemented the *CCapeThermoPropertyPackage* class so that it provides a method for the graphical user interface to obtain a reference to its data table. This allows the user interface to include functionality to edit the data table and to link it to external sources, such as a Microsoft Excel spreadsheet or a SQL database server. The information provided by a Property Package can be presented rather concisely, as shown in Figure 2.28.

Also attached to the *CCapeThermoMaterialObject* is the *CCapeReactionChemistry* class, which is a chemical reaction database based on the pertinent CAPE-OPEN chemical reactions interface specification (CO-LaN 2003h). The *CCapeReactionChemistry* class consists of a collection of chemical reactions, which in turn consist of collections of reactants. According to Barrett and Yang, a reactant can be implemented as a data structure that has the following members: an integer reaction coefficient, a string that contains the name of the chemical species, an integer that contains the ionic charge of the chemical species, a string containing the Chemical Abstract Services (CAS) number of the chemical species, a Boolean value indicating whether the reactant is the base reactant of the reaction, and a string indicating the phase in which the reaction occurs. The *CCapeReactionChemistry* class contains methods that return information about a specific reaction. For example, the *CCapeReactionChemistry.GetReactionCompoundIds* method returns the names of the chemical species that are involved in the specified reaction.

Property Packages

CAPE-OPEN specifications include definitions for different types of thermodynamic calculation components. It is possible to write software components that only know how to calculate selected properties, or that only know how to calculate thermodynamic phase equilibrium. However, the only thermodynamic software component that is widely supported is a property package, the concept of which is illustrated in Figure 2.28. (van Baten and Pons 2014)

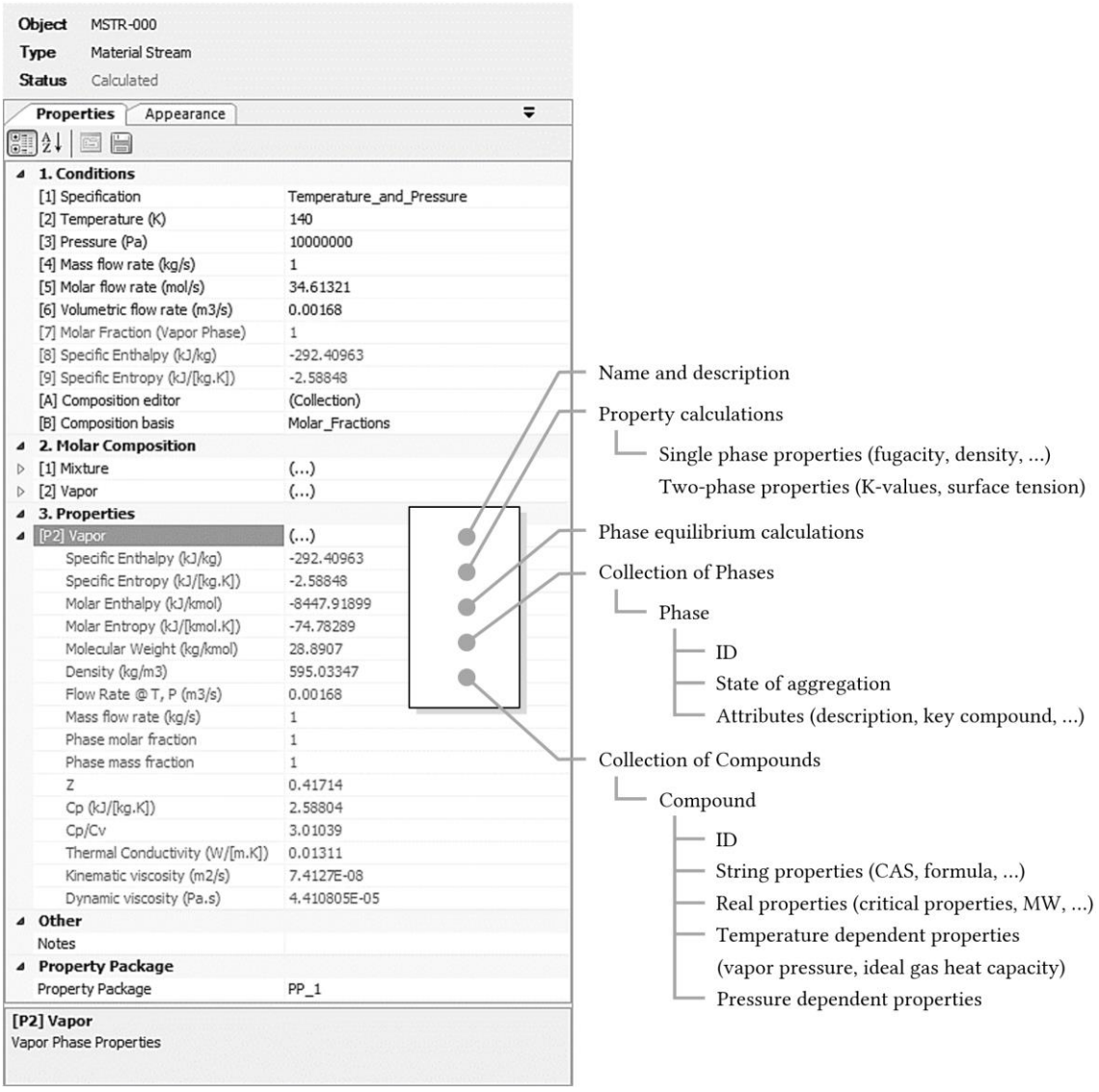


Figure 2.28 Functionality exposed by a CAPE-OPEN property package (adapted from van Baten and Pons 2014; Medeiros et al. 2014b).

A property package defines compounds and phases, and includes the calculation methods for thermodynamic properties and thermodynamic phase equilibria. In fact, a material type in a flow sheet simulation is fully configured only when it is associated with a property package.

However, the PME can make further configurations; for example, it may choose to use only a subset of compounds defined by a property package. Typically, a property package is preconfigured using separate configuration software, and all that is left to the PME is to instantiate and access the property package. It is common practice to group these configurations together in a Thermodynamic System, or a Property Package Manager. Such a manager can be used by a PME to request a list of all available property packages, and subsequently to instantiate a property package from that list (as shown in Figure 2.17 on page 20). (van Baten and Pons 2014)

The calculation conditions, including the temperature, pressure and composition, are stored on a material object. The material object is passed to the property package for property and phase equilibrium calculations. The results for the calculations are stored on the material object, which may or may not be the same material object that is associated to a material stream connected to the port of a unit operation. Material objects expose the same interfaces as property packages, and consequently a unit operation has access to all the functionality that is exposed by a property package, even if a CAPE-OPEN property package is not in fact used but the calculations are performed by the PME itself. (van Baten and Pons 2014)

The property package (PP_1) referred to in Figures 2.17 and 2.28 is a property package that comes with the DWSIM simulator, which is introduced in Section 2.4.1. It uses the Peng-Robinson equation of state (EOS) for some of its property calculations, as shown in Table 2.6.

Table 2.6. Methods and correlations used by the PP_1 Property Package (Medeiros et al. 2015a)

Phase	Fugacity	H / S / C_p / C_v	k^a	Viscosity	Density
Vapour	EOS	EOS	Ely-Hanley	Lucas / JST[b]	EOS
Liquid	EOS	EOS	Latini	Letsou-Stiel	EOS / Rackett

[a] Thermal conductivity, [b] Jossi-Stiel-Thodos

The Peng-Robinson equation of state (Peng and Robinson 1976) is an improvement on the ideal gas law, the Van der Waals equation of state (1873) and others. It is a cubic equation of state (the molar volume is squared), which relates the temperature, pressure and molar volume of a pure component or a mixture of components at equilibrium. In point of fact, cubic equations are the simplest equations capable of representing the behaviour of liquid and vapour phases simultaneously. The Peng–Robinson equation of state can be written in the following form

$$p = \frac{RT}{V_m - b} - \frac{a(T)}{V_m^2 + 2bV_m - b^2} \tag{2.1}$$

where p is pressure, R is the ideal gas constant, V_m is molar volume, b is the parameter related to hard-sphere volume and $a(T)$ is the temperature-dependent parameter related to intermolecular forces. The values for R, b and a can be found in the literature. (Medeiros et al. 2015b).

2.3.6 Extensions, add-ins and prototyping

This section introduces a selection of extensions and utilities wrapped as unit operations that have been developed for CAPE-OPEN compliant simulators. Also, some of them present mechanisms that are used for handling systems of equations inherent to both steady-state and equation–oriented simulation. The examples herein show how to create completely new models without the need to write code, and how to utilize general-purpose modelling tools.

Incorporation of computational tools

Morales-Rodríguez et al. (2008) have shown how to use the CAPE-OPEN interfaces to enable interoperability between a modelling tool and a process simulator. Their modelling tool of choice was ICAS-MoT, see (CAPEC 2016a), and they used ProSimPlus, see (ProSim 2016), as the process simulator. The unit operation they created was a direct methanol fuel cell (DMFC).

DMFCs are used in the industrial production of methanol. A flow diagram for a DMFC modelled though the multi-scale approach is shown in Figure 2.29. Multi-scale modelling basically consists of the division of a complex model into a set of sub-models that are described on different length or time scales in order to obtain a more accurate model.

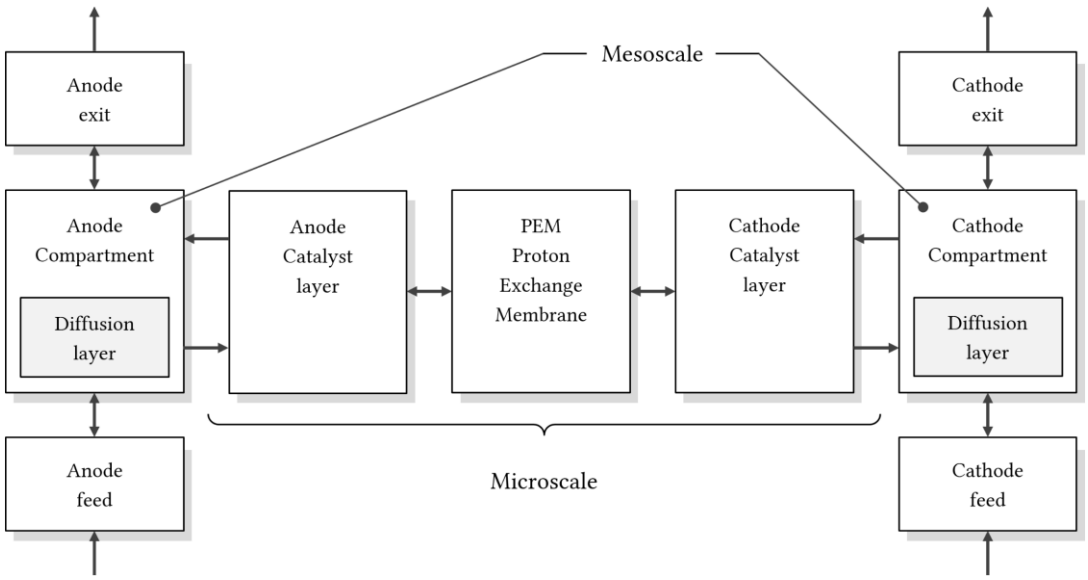


Figure 2.29 Direct methanol fuel cell through the multi-scale approach (adapted from Morales-Rodríguez 2008).

Two different scales are involved: meso-scale and micro-scale. Meso-scale includes the modelling of anode and cathode compartments while micro-scale is employed to model the behaviour of anode and cathode catalyst layers and the proton membrane exchange. A subset of equations that model the DMFC unit operation is shown in Table 2.7, which also includes a description for each type of equation, and their respective scales within the unit operation.

Morales-Rodríguez et al. included two modelling scenarios in their study, namely multi-scale and single-scale. In the multi-scale modelling approach the entire set of equations was solved. For the single-scale scenario, only the equations at meso-scale, Eq. 1 and 2 in Table 2.7, were solved, and the variables in the equations defining micro-scale were given known, condition-dependent values. The values had been acquired from a prior multi-scale steady-state simulation. Alternatively, they could have been obtained by looking up data from the literature.

Table 2.7. DMFC model equations (Morales-Rodríguez et al. 2008)

Scale	Type	Description	Equation	
Meso-scale	Mass balance	Anode compartment	$\frac{dc_{CH_3OH}}{dt} = \frac{1}{\tau}(c^F_{CH_3OH} - c_{CH_3OH}) - \frac{k^{LS}A^S}{V_a}(c_{CH_3OH} - c^{CL}_{CH_3OH})$	(1)
			$\frac{dc_{CO_2}}{dt} = \frac{1}{\tau}(c^F_{CO_2} - c_{CO_2}) - \frac{k^{LS}A^S}{V_a}(c_{CO_2} - c^{CL}_{CO_2})$	(2)
			⋮	
Micro-scale	Constitutive equations	Electrode kinetics	$r_5 = k_5 exp\left(\frac{\alpha_5 F}{RT}\eta_a\right)\left\{1 - exp\left(-\frac{F}{RT}\eta_c\right)\left(\frac{p_{O_2}}{p^0}\right)^{3/2}\right\}$	(9)

Methanol and carbon dioxide bulk concentrations, i.e. the variables at the meso-scale level, were chosen to illustrate the results obtained for both the multi-scale (MS) and the single-scale (non-MS) model scenarios. The results of the simulations are shown in Fig. 2.30.

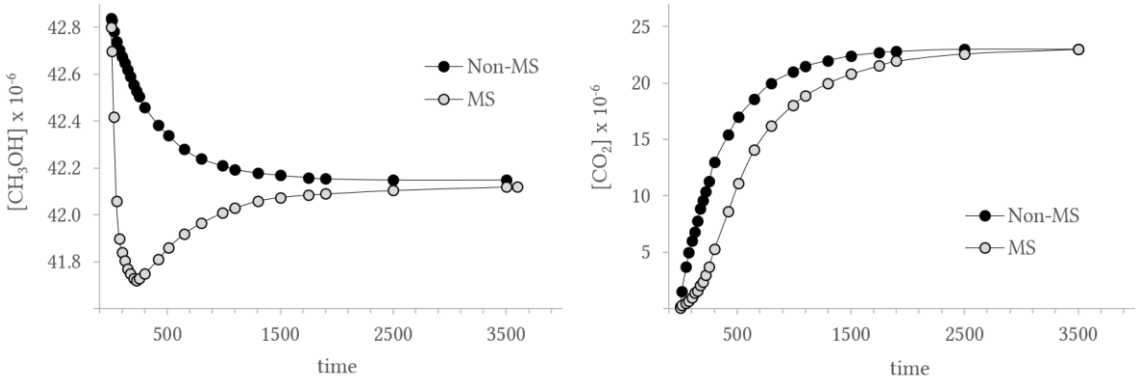


Figure 2.30 Time-dependent concentrations of CH_3OH (left) and CO_2 (right).
The results were adapted from Morales-Rodríguez (2008) and redrawn using Microsoft Excel.

Fig. 2.30 makes the differences between these two models clear, as the concentrations change differently with respect to time at the beginning of the simulation, but are almost equal at steady state – when enough time has lapsed (the original plots contained no information regarding the units of measure). Clearly, the multi-scale model provides a gain in accuracy.

Figure 2.31 shows an arrangement of objects within the DMFC unit operation. The equations representing the model were stored in an ICAS-MoT file, which was accessed by an ICAS-MoT object (a COM object). The use of CAPE-OPEN interfaces made the whole entity CAPE-OPEN compliant. In addition, an XML configuration file was used for describing the mapping between variables in the ICAS-MoT model and the variables in the CAPE-OPEN data model.

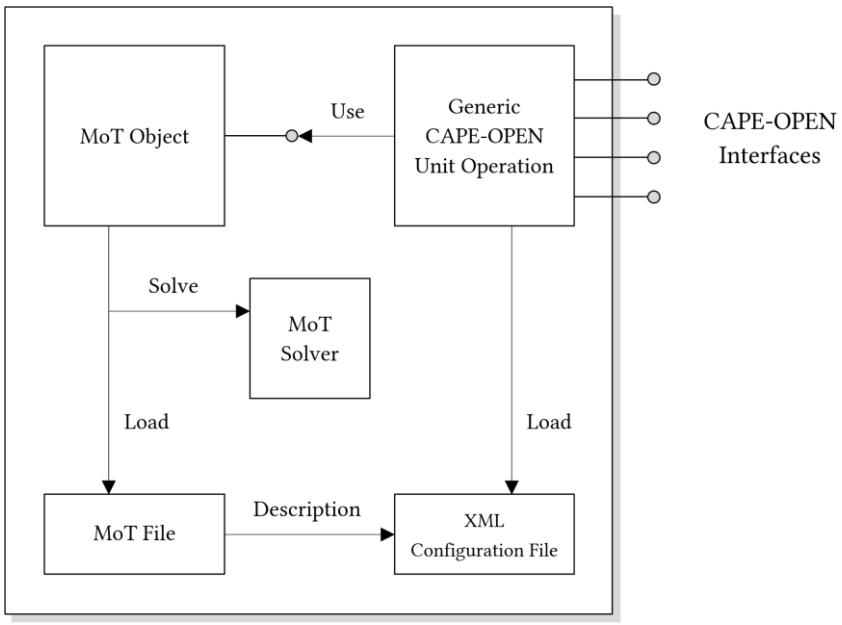


Figure 2.31 Generic CAPE-OPEN unit operation (adapted from Morales-Rodríguez et al. 2008).

The input Material Objects must provide information concerning their temperature, pressure and composition, which are expressed as either total flow rate and molar fractions, or partial flow rates. These were written into the XML configuration file. Consequently, all the values of the variables were transferred to the MoT file, and utilized in the calculations by the MoT Solver. Once the solution was obtained, the results were written into an output Material Object. The results included the temperature, pressure, composition and enthalpy for the product stream. In this case, the enthalpy was calculated by ProSimPlus instead of the MoT Solver.

Interestingly, ProSimPlus is not a dynamic simulator, but the DMFC model contained in the ICAS-MoT file is *dynamic* (the modelled phenomenon is time-dependent, as the results in Fig. 2.30 indicate). Moreover, the range of problems that the ICAS Solver can handle becomes interesting, too, in the case where there is a plan to harness its power. According to (CAPEC 2016b), it has solvers for at least AEs, ODEs, DAEs, PDEs, in addition to optimization capabilities (LP, NLP, MILP and MINLP). The meanings of the abbreviations are listed on pages v–vi.

COWAR unit operation

The evaluation of the sustainability of a chemical process can be performed using qualitative and quantitative indicators. Quantitative indicators, such as potential environmental impact (PEI) indexes, are essential for evaluating the environmental impact of a chemical process, and also for choosing the best design among available alternatives. The Waste Reduction (WAR) Algorithm (Cabezas et al. 1997; Young and Cabezas 1999; Young et al. 2000) used for calculating PEI indexes, has been implemented by Fermeglia et al. (2008) using CAPE-OPEN standards. COWAR, their unit operation-like utility, is designed to evaluate process sustainability.

The WAR algorithm equations are located in the unit operation's *Calculate* method. The *Calculate* method, defined in the *ICapeUnit* interface, is also connected to an Environmental Protection Agency (EPA) database, which contains the environmental impact categories for more than 1700 substances. The unit operation makes the output stream equal to the input stream, so that the flow rate, temperature, pressure and composition stay unchanged. The database is queried for the normalized values for the environmental impact categories of the substances involved in a chemical process. Relevant environmental impact indexes are calculated using the PEI balance equations for the chemical process and energy usage.

The overall potential environmental impact of chemical k, Ψ_k, can be determined by summing up the specific potential environmental impacts of chemical k, $\Psi^s_{k,l}$, in all the impact categories

$$\Psi_k = \sum_l \alpha_l \Psi^s_{k,l} \tag{2.2}$$

where α_l represents the relative weighting factor of impact category l. The weighting factors are essential for the WAR algorithm because they permit a change in the relative importance of the impact categories according to local needs and policies. (Fermeglia et al. 2008)

The WAR algorithm uses eight different environmental impact categories in its evaluation. The categories include human toxicity potential by ingestion, human toxicity potential by exposure (both dermal and inhalation), terrestrial toxicity potential, aquatic toxicity potential, global warming potential, ozone depletion potential, photochemical oxidation potential and acidification potential. (Young et al. 2000)

Flowsheet monitoring

Flowsheet monitoring is the ability to access all the elements present in a flowsheet without interfering with it. Flowsheet monitoring components are software components that can be plugged into a flowsheeting tool and have read-only access to all thermodynamic property calculation methods, all streams and all unit operations. (van Baten 2008a)

Because there had been no suitable CAPE-OPEN interface available that could have been used in flowsheet monitoring applications, a proposal for an appropriate extension was made and presented in (van Baten 2008a, 2008b; Barrett et al. 2011). Generally, the applications that would use flowsheet monitoring perform post-processing on flowsheet calculations, such as flowsheet evaluation and validation. The COCO simulator introduced in Section 2.4.1 comes with two add-ins which utilize flowsheet monitoring: TERNYP and WAR.

WAR add-in

Barrett et al. (2011) describe how the WAR algorithm was implemented as a process simulation add-in utilizing the flowsheet monitoring discussed above. The WAR add-in accesses the chemical flows and unit operations directly from the process simulator and calculates the WAR PEI score for the modelled process using process data obtained directly from the flowsheet. Users can select either the entire flowsheet or a sub-graph of individual unit operations for the WAR evaluation. Additionally, the WAR option of including impacts resulting from energy use has been made available. The WAR add-in has been tested and validated against the flowsheet monitoring interface in the COFE process-modelling environment that is shipped with COCO, see Section 2.4.1. (Barrett et al. 2011)

Barrett et al. chose this approach over COWAR, since the approach COWAR uses is inflexible, in that if the user wants to select a particular subset of the flowsheet, they would need to add and/or remove unit blocks, which would effectively invalidate the flowsheet, thus requiring its recalculation. Furthermore, the addition of units to the flowsheet would convolute the actual flow diagram, and increase the calculation time by increasing the number of flash (i.e. equilibrium) calculations that need to be performed. This is because the upstream unit must perform one flash calculation on the inlet stream to the COWAR block, which then requests that the outlet material stream performs a second flash calculation. (Barrett et al. 2011)

The WAR algorithm has been implemented in a number of products. For example, the AspenPlus and ChemCAD commercial simulators have the WAR algorithm added to them, and it has also been implemented as part of the ICAS chemical process design software. A list of relevant primary references is included in (Barrett et al. 2011).

Gibbs reactor

A description of how to create a single-phase equilibrium reactor model is given in (van Baten and Szczepanski 2011). The reactor model calculates the reaction equilibrium at a fixed pressure by minimization of the Gibbs free energy at constant temperature, or maximization of entropy at constant enthalpy. In short, the problem can be formulated as

$$\min G(\xi) \text{ or } \min -S(\xi) \tag{2.3}$$

where ξ is the reaction extent and appropriate non-negativity constraints concerning mole fractions and feed stream compositions apply (not presented here). The user is allowed to specify the reactions explicitly, or to make a selection of the compounds involved in the reaction, which can be all or a subset of the compounds that are present in the feed Material Object. Additionally, techniques for an automatic procedure for determining the reactions and stoichiometry are presented.

The reactor model has been implemented in compliance with the CAPE-OPEN standards, and can therefore run in multiple simulation environments using different thermodynamic engines. The possibilities and limitations of CAPE-OPEN are clear. For example, it is assumed that first-order derivatives of all property calculations with respect to pressure, temperature and composition are available, but that second-order derivatives are not.

The modelling of a Gibbs reactor is essentially an optimization problem. Specific care must be taken by the unit operation not to evaluate thermodynamic properties in conditions for which the thermodynamic server utilized in the calculations may not provide answers, and it is important to evaluate thermodynamic properties only at mole fractions in the [0, 1] region.

Additionally, it is important to make sure that none of the interim solutions violate any constraints. All the solutions have to lie in a feasible area, which in this case is a convex hyperplane bounded by the non-negativity constraints. A projection method was applied to ensure that the solutions were within the feasible area. Further information on creating simulation and optimization algorithms is also provided in (Morton 2003).

Prototyping with Scilab, MATLAB and Excel

A substantial amount of knowledge of CAPE-OPEN interfaces is required before one can commence implementation of a CAPE-OPEN unit operation. Additional know-how requirements for a developer come from the use of COM as the middleware. (van Baten 2009)

It is possible that the developer possesses a legacy version of the unit operation model that is already available, implemented for example with Microsoft Excel, Scilab or MATLAB. These modelling environments have the advantage of being intuitive to use, and they come with a number of utilities that the model developer can use to solve the model equations. One example of such a utility is the Excel Solver add-in. It would help to have an option to prototype the model (i.e. build a working version of the model for testing whether the model performs as intended) using these generic modelling tools, before creating a final CAPE-OPEN based unit operation implementation. (van Baten 2009)

Often the development of a process model is performed outside the context of the simulation environment that it is intended to run in. All that is required to develop such a process model is access to the thermodynamic and physical property calculations and flash calculations. In fact, AmsterCHEM has developed tools that enable this within MATLAB; for more information see http://www.amsterchem.com/matlabunitop.html, and also within Scilab: for more information see http://www.amsterchem.com/scilabunitop.html. Besides having the modelling environment installed, one should have an installation of the relevant, MATLAB or Scilab thermodynamic import tools and a valid CAPE-OPEN thermodynamic server. A number of such servers are available, such as the TEA server that is shipped with COCO, see Section 2.4.1. Also, some companies may have their own in-house thermodynamic servers available via CAPE-OPEN. (van Baten 2009)

Hence, access to thermodynamic calculations in MATLAB or Scilab models no longer requires any knowledge of CAPE-OPEN programming. Ready-made functions exist for the creation and maintenance of CAPE-OPEN thermodynamic property packages. Upon demand, a so-called property package handle is returned that can be used for further calculations. For example, in MATLAB, one can create a “C1_C2” property package of the TEA thermodynamic server, containing methane and ethane, by issuing the first two commands shown in Fig. 2.32.

Functions are also available for temperature-dependent property calculations, single-phase mixture property calculations, and two-phase mixture property calculations. An example calculation of vapour enthalpy at equimolar composition, 10^5 Pa pressure and a range of tem-

perature is shown in Figure 2.32 (the last command). The commands can be issued from any script containing the model equations, or from the MATLAB or Scilab command line, thus enabling script-based and interactive model development. (van Baten 2009)

```
» handle=capeOpenGetPackage('TEA (CAPE-OPEN 1.1)','C1_C2');
» capeOpenCompounds(handle)

ans =

 'Methane'
 'Ethane'

» capeOpen1PhaseProp(handle,'enthalpy','vapor',[300:20:400]',1e5,[0.5 0.5])'

ans =

 1.0e+003 *

 0.0467  0.9525  1.8909  2.8631  3.8699  4.9123
```

Figure 2.32 Thermodynamic functions used within MATLAB (adapted from van Baten 2009)

After creating the model with a modelling tool, the developed model is inserted into a flowsheet simulation environment as a CAPE-OPEN unit operation. Instead of working with thermodynamic servers that are explicitly loaded, the underlying thermodynamic services of the simulation environment are used. The services can be CAPE-OPEN based or native to the simulation environment. However, before commencing this phase, an installation of a CAPE-OPEN compliant simulation environment is required, along with the MATLAB or Scilab programs, and the unit operation tools. (van Baten 2009)

Definition of a unit operation starts by defining its feed and product ports and input and output parameters. This is done by using the unit operation configuration window, which is accessed by starting the unit operations editing procedure. If the unit operation produces textual reports, these can be configured, too. The output of the calculation is made available as a textual report by default. The actual script that calculates the model can be entered into the script-editing panel which is part of the configuration window. (van Baten 2009)

Prototyping in Excel is somewhat different than the script-based prototyping of MATLAB and Scilab. Instead, the model equations are entered in the traditional Excel way by entering calculation formulas that have references to cells which contain the input data. The Excel Unit Operation does not have a dedicated GUI. When editing, the Excel application will launch. Within Excel, the *Feeds* and *Products* worksheets contain tables of feeds and products and their properties. Adding and removing feeds or products is done by adding or deleting rows to/from these tables. The pressure, temperature, composition and enthalpy for each of the feeds are listed on the *Feeds* worksheet. The modeller has to enter formulas that provide values for each product stream, for composition and for two of the following three state variables: pressure, temperature and enthalpy. Similarly, input and output parameters are configured by making changes to the parameter tables within the relevant worksheets. The *Control* worksheet contains facilities for allowing the specification of solution status, returning solution error messages and configuring the Excel Solver add-in to solve sets of equations. (van Baten 2009)

Equation Set Objects

Whether the simulation is equation-oriented or sequential modular, the equations describing the whole process or individual unit operations are aggregated into a system of *n* non-linear equations in *n* unknowns given by

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{2.4}$$

Rewriting the system in a single expression using vectors gives

$$f(x) = 0 \tag{2.5}$$

where the vector x contains all the variables associated with the flowsheet in a dynamic simulation, or the variables associated with an individual unit operation in a sequential modular simulation. These variables include input variables, such as feed stream data, output variables such as product stream data, and internal variables that are not visible to the PME (CO-LaN 2006). The vectors x, and f that contains the functions $f_i(x)$ can be written

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, f(x) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) \end{bmatrix} = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} \tag{2.6}$$

These kinds of systems of equations are often solved using the Newton-Raphson method or some variant of it (Morton 2002), which require the evaluation of a matrix, known as the Jacobian of the system. The Jacobian is defined as

$$\mathcal{J} = \begin{bmatrix} \partial f_1 / \partial x_1 & \partial f_1 / \partial x_2 & \cdots & \partial f_1 / \partial x_n \\ \partial f_2 / \partial x_1 & \partial f_2 / \partial x_2 & \cdots & \partial f_2 / \partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n / \partial x_1 & \partial f_n / \partial x_2 & \cdots & \partial f_n / \partial x_n \end{bmatrix} \tag{2.7}$$

An initial guess for the solution $x = x_0$ is required. Successive approximations to the solution are obtained from

$$x_{n+1} = x_n - \mathcal{J}^{-1} \cdot f(x_n) = x_n - \Delta x_n \tag{2.8}$$

with $\Delta x_n = x_{n+1} - x_n$.

Several possible convergence criteria for the solution of a system of non-linear equations exist. One of them is that the maximum of the absolute values of the functions $f_i(x_n)$, the so-called l_∞ norm, is smaller than a certain tolerance ε

$$\max_i \left| f_i(x_n) \right| < \varepsilon \tag{2.9}$$

Another possibility for convergence is that the magnitude of the vector $f(x_n)$, the l_2 norm, also known as the Euclidean distance, has to be smaller than the tolerance ε

$$|f(x_n)| < \varepsilon \tag{2.10}$$

The difference between consecutive values of the solution, which is the length of the Newton step, can also be used as convergence criteria

$$|\Delta x_n| = |x_{n+1} - x_n| < \varepsilon \tag{2.11}$$

One of the main complications with using Newton to solve a system of non-linear equations is the need to define all the functions $\partial f_i / \partial x_j$, for $i, j = 1, 2, \ldots, n$, included in the Jacobian. Clearly, as the number of equations and unknowns, n, increases, so does the number of elements in the Jacobian, n^2.

Also, sometimes it is the case that the partial derivatives are not obtainable for one reason or another. One way to circumvent this problem is to use the secant method, in which the Jacobian is approximated through finite differences, as shown in the following equation

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(x_1, x_2, \ldots, x_j + \Delta x, \ldots, x_n) - f_i(x_1, x_2, \ldots, x_j, \ldots, x_n)}{\Delta x} \tag{2.12}$$

where Δx is a small increment in the independent variables. Note that $\partial f_i / \partial x_j$ represents the element $\mathcal{J}_{ij}$ in the Jacobian $\mathcal{J} = \partial(f_1, f_2, \ldots, f_n) / \partial(x_1, x_2, \ldots, x_n)$.

The basis for the above discussion on solutions of non-linear equations can be found in (Urroz 2004). Further to the general method presented above, a project report from Barrett et al. (2007) describes the use of the Jacobian for solving particular aqueous equilibrium calculations in a self-made steady-state simulation environment. Their system consists of a Watts plating bath in which the equations f_i are the equations defining total concentrations of different components present in the bath, and the variables x_i are the concentrations of the compounds.

One of the components is boric acid, H_3BO_3. Some boric acid is fed directly into the bath, besides which the total boric acid concentration is affected by the following two reactions:

$$H^+ + B(OH)_4^- \leftrightarrow H_3BO_3 + H_2O \tag{2.13}$$

$$H^+ + B_3O_3(OH)_4^- + 2\,H_2O \longleftrightarrow 3\,H_3BO_3 \tag{2.14}$$

These reactions reach an equilibrium state and the total concentration of boric acid is obtained from the following equation (kinetics are neglected, as the reactions are assumed to be fast)

$$[H_3BO_3]_{total} = [H_3BO_3] + \frac{[H_3BO_3][H_2O]}{[H^+]K_{B(OH)_4^-}} + 3\frac{[H_3BO_3]^3}{[H^+][H_2O]^2K_{B_3O_3(OH)_4^-}} \tag{2.15}$$

All the reactions that take place in the plating bath are described with similar equations that define the total concentrations. The partial derivatives for these equations can be found by assuming that the concentrations of all the components are constant, except the component of interest, and by using the power rule to obtain the derivatives. As an example, the derivative of the concentration of boric acid with respect to [H+] is simply

$$\frac{\partial[H_3BO_3]_{total}}{\partial[H^+]} = -\frac{[H_3BO_3][H_2O]}{[H^+]^2K_{B(OH)_4^-}} - 3\frac{[H_3BO_3]^3}{[H^+]^2[H_2O]^2K_{B_3O_3(OH)_4^-}} \tag{2.16}$$

Note that the equilibrium constant K in equations 2.15 and 2.16 is temperature-dependent. Unless an assumption is made that the temperature in the bath is constant, it has to be evaluated somehow. Eventually, a Jacobian matrix can be formed by aggregating all the partial derivatives, and the solution is calculated iteratively by the use of equation 2.8 until a predefined convergence criterion has been met.

Some of the mathematics presented above can be handled with the use of the Equation Set Object (ESO), which is part of the CAPE-OPEN standards, see (CO-LaN 1999a). CAPE-OPEN also provides interfaces for numerical solvers (CO-LaN 1999b) that have been designed with the intention of meeting the demands of solving large, sparse systems of non-linear algebraic equations (NLAEs) and mixed (ordinary) differential-algebraic equations (DAEs).

The ESO allows the description of a set of equations

$$f(x, \dot{x}, t) = 0 \tag{2.17}$$

and provides the methods for example for obtaining the number of variables and the number of equations, obtaining and setting the values of x, $\dot{x}$ and t, obtaining the values of functions and the number of non-zero derivatives, obtaining the structure of the partial derivatives (the Jacobian), and obtaining and setting the values of the derivatives.

An examination of the CAPE-OPEN numerical interfaces by Soares and Secchi (2003) revealed that there is an efficiency loss in using the interfaces, and their use is not recommended when dealing with large systems of equations. They also proposed modifications to the interface set, and ran usability tests with the modified set of interfaces. Yang and Biegler (2005) describe the

development of CAPE-OPEN compliant objects used with their large-scale nonlinear programming solver. Also included in (Yang and Biegler 2005) is a description of the use of ESOs to allow interfacing to different modelling systems. Furthermore, Thomas (2011) mentions ESOs in his paper on dynamic simulation as a technique that might be utilized in the future development of unit operations for particular dynamic simulation environments.

A library for equation system processing based on the CAPE-OPEN ESO interface has been devised by Schopfer et al. (2005), called *LptNumerics*. It provides reusable functionality that is applicable in different equation-oriented modelling and simulation tools. However, according to van Baten and Pons (2014), there are not many (other) implementations of CAPE-OPEN numerical solvers. To this end, *LptNumerics* library serves as a proof of concept, and provides information on ways to design CAPE-OPEN compliant numerical solvers.

If the use of ESOs and the solvers that utilize them is not seen as a viable option, the development of proprietary mathematical model solvers may be considered. Requirements for the development of solvers include the pertinent mathematical techniques, and an understanding of the properties of floating point numbers, which set a certain resolution to the calculations.

If equation 2.6 was assumed to consist of linear equations, it can be written

$$Ax = b \tag{2.18}$$

The computed solution x_* of a linear system of equations almost always differs somewhat from the theoretical solution $x = A^{-1}b$, because of rounding errors. If the elements of x are not floating-point numbers (for example, any multiple of pi), then x_* cannot equal x. (Moler 2004). There are two common measures of discrepancy in x_*, the error

$$e = x - x_* \tag{2.19}$$

and the residual

$$r = b - Ax_* \tag{2.20}$$

Moler (2004) shows that while the residual is small, the error may in fact be quite large. As some applications use residuals to estimate the nearness to the theoretical solution, caution and further verification of the correctness of solution are still required.

Besides the rounding error, truncation errors also produce erroneous results. For example, Adams and Essex (2010) illustrate numerous examples of how these types of error manifest themselves in calculations carried out by computers. A major issue is the situation where an initially non-singular matrix becomes singular due to rounding and/or truncation errors. Equally, a singular matrix can unintentionally become non-singular. Consequently, if the Jacobian becomes accidentally singular, techniques involving equation 2.8 cannot be directly employed. These are rather common problems and techniques have been devised to solve them, see for example (Morton 2003) and (Moler 2004).

2.4 Availability of compliant products

The clearest evidence of the viability of CAPE-OPEN standard specifications comes in the form of simulation environments that comply completely or partially with the CAPE-OPEN standards. Two non-commercial simulation environments, COCO and DWSIM, are introduced here. Several commercial simulation environments exist, and a recently written review of them is available (van Baten and Pons 2014), with some of its content presented in Sec. 2.4.2.

2.4.1 COCO and DWSIM

COCO (CAPE-OPEN to CAPE-OPEN) is a free-of-charge CAPE-OPEN compliant steady-state simulation environment that is used for simulation, and also for CAPE-OPEN compliancy testing. It comes with a number of relevant utilities, see Table 2.8, and a collection of unit operations and auxiliary unit operation-like tools that are listed overleaf. (van Baten et al. 2015)

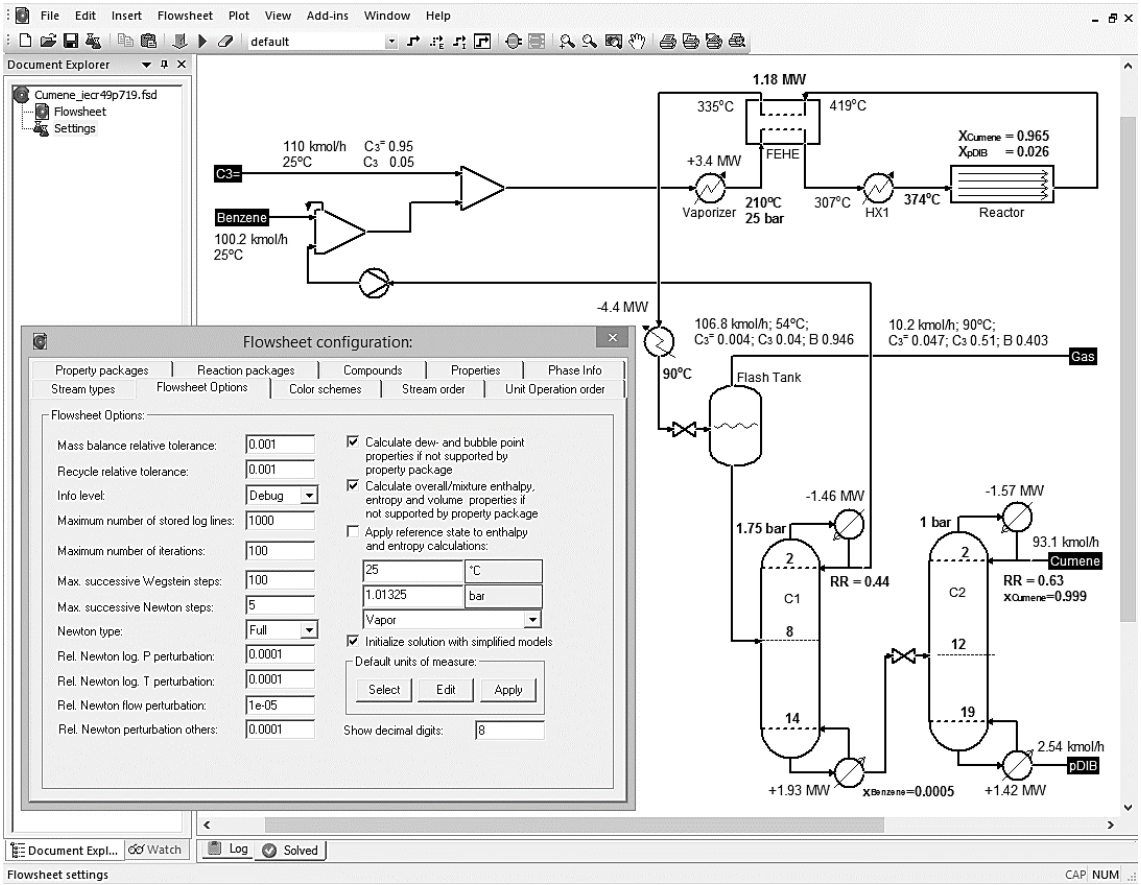


Figure 2.33 COFE with a sample flowsheet loaded, and the settings dialog box open.

COCO consists of the graphical flowsheeting tool COFE (CAPE-OPEN Flowsheet Environment), shown in Figure 2.33, property package TEA (Thermodynamics for Engineering Appli-

cations), unit operation collection COUSCOUS (CAPE-OPEN Unit operations Simple) and reaction package CORN (CAPE-OPEN Reaction Numeric package). It is possible to find information concerning the development of CAPE-OPEN components, as well as several sample flowsheets for download that can be loaded into COFE at www.cocosimulator.org. Additionally, the manual and documentation for COFE and its utilities are available on the same website.

Table 2.8. COCO utilities and add-ins (van Baten et al. 2015)

Utility	Remark
COFE.xlt	A Microsoft Excel spreadsheet template for embedding a COFE document in an Excel workbook. Allows for access to stream and unit operation data, and performing of thermodynamic and property calculations in Excel.
ConfigureCORN	Used for configuring and maintaining CORN reaction packages.
ConfigureTEA	Used for configuring and maintaining TEA property packages.
CORK	CAPE-OPEN Registry Kit. A tool for inspecting CAPE-OPEN component registration information in the Windows registry.
CUP	A utility for updating all packages included in COCO.
JUIcE	An icon editor for creating unit operation icons to be used in COFE.
OATS	Allows an out-of-proc run of property packages (COM). Includes facilities for logging CAPE-OPEN communication between a PME and PMCs. Also includes COULIS, a Unit Operation Logger utility.
Online help	Online documentation.
TERNYP	Used for creating phase diagrams, property plots and residue curves of ternary systems. (TERNarY Plugin).
Water	Stand-alone property package for calculation of properties of water and steam. Conforms to CAPE-OPEN thermodynamics version 1.1.

The collection of unit operation and unit operation-like auxiliary tools that come with COUSCOUS include an expander, pump, heater, cooler, compound splitter, plug flow reactor (PFR), stream converter, mixer, splitter, valve, Gibbs reactor, heat exchanger, equilibrium reactor, fixed conversion reactor, flash, continuously stirred tank reactor (CSTR), void operation, 3-phase flash, turbine, compressor, solid separator, property tester, information calculator,

measure unit, heat of combustion unit, thermal energy mixer, thermal energy splitter, COULIS logger, embedded flow sheet, Excel model, MATLAB or Scilab model, ChemSep column model, controller and flow constraint. It should be noted that a complete COFE flow sheet can be used as a CAPE-OPEN compliant unit operation for use inside COFE or other simulators. (van Baten et al. 2015)

Another free-of-charge CAPE-OPEN compliant simulation environment introduced here is the DWSIM (Medeiros et al. 2015c), shown in Fig. 2.34. It is open-source, written in Visual Basic and C#, and available for Windows and Linux. DWSIM targets .NET Framework 4.0.

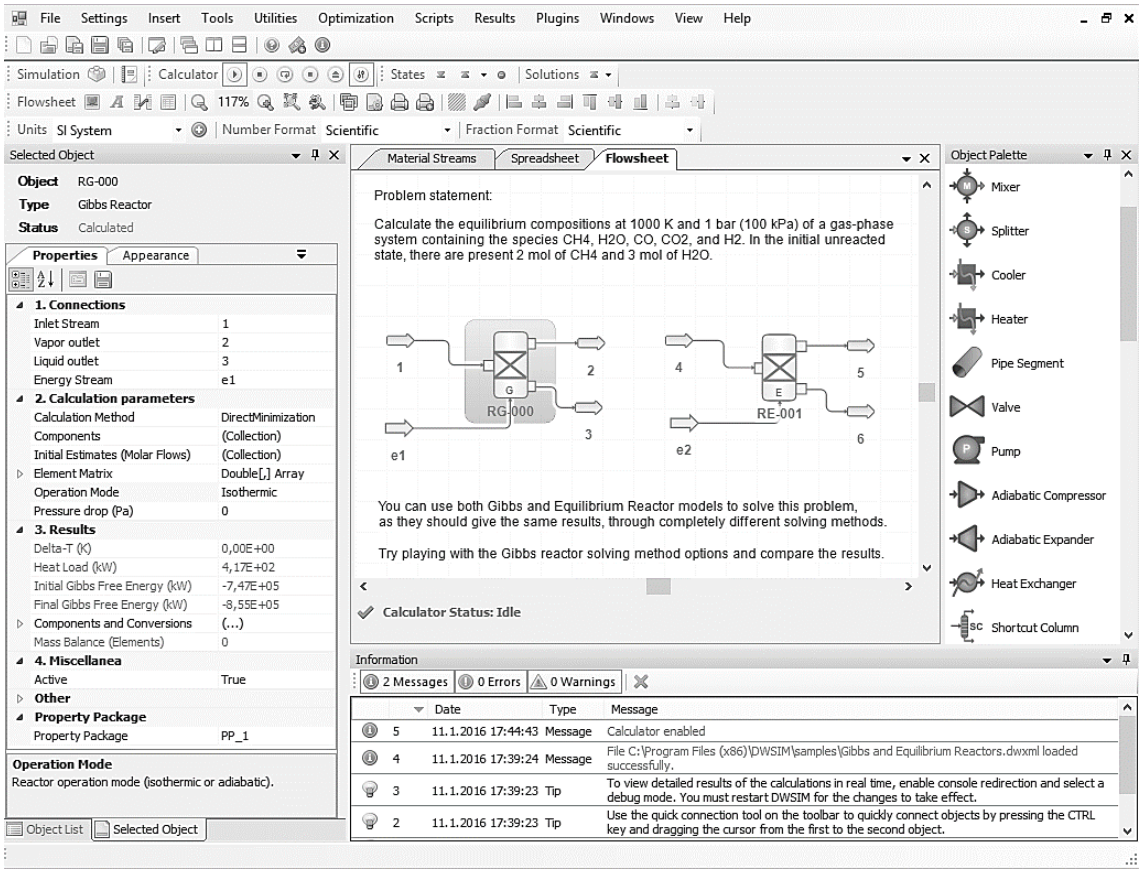


Figure 2.34 Screenshot of DWSIM after a sample flowsheet has been loaded.

DWSIM features are presented in Table 2.9 and the unit operations included are listed below Table 2.9. Information on DWSIM, such as its manual, is available at dwsim.inforside.com.br, where there is also documentation on CAPE-OPEN component development. At the time of writing, all the source code files for DWSIM are available for download from either GitHub, github.com/DanWBR/dwsim3, or SourceForge, sourceforge.net/projects/dwsim/.

According to the statistics on SourceForge, the source code had been downloaded more than 26 000 times in 2015. Downloads had been initiated from a total of 141 countries and 73 % of

downloaders had used Windows. In addition, the thermodynamic calculation engine used in DWSIM has been made available as a standalone DLL file that can be used in other simulation applications. In fact, it serves as the basis for a mobile device application DWSIM Calculator that calculates the phase equilibria and properties of mixtures of substances.

DWSIM comes with a comprehensive set of flowsheet templates that help users to start using the software. The modelling of custom unit operations is done through the use of scripting languages IronPython and IronRuby, see also Section 3.1.2. For up-to-date information about the features in Table 2.9, the reader is advised to visit the DWSIM website.

Table 2.9. DWSIM features (Medeiros et al. 2015c)

Feature	Remark
CAPE-OPEN	Thermo 1.0/1.1 Property Package Socket, Thermo 1.1 Property Package Server, Unit Operation Socket and Flowsheet Monitoring Object support. DWSIM also exposes an IronPython/IronRuby Script (Custom) Unit Operation for CAPE-OPEN compliant simulators.
Thermodynamic models	PC-SAFT, FPROPS, CoolProp, Peng-Robinson, Soave-Redlich-Kwong, Lee-Kesler, Lee-Kesler-Plöcker, UNIFAC, Modified UNIFAC (Dortmund), UNIQUAC, NTRL, COSMO-SAC, LIQUAC, Extended UNIQUAC, Chao-Seader, Grayson-Streed, Raoult's law, IAPWS-IF97 steam tables, IAPWS-80 Seawater, Black-Oil and Sour Water.
Utilities	Phase Envelope, Hydrate Calculations, Pure Component Properties, Critical Point, PSV Sizing, Vessel Sizing, Spreadsheet and Petroleum Cold Flow Properties.
Tools	Binary Data Regression, Compound Creator, Bulk C7+ and Distillation Curves Petroleum Characterization and Reactions Manager.
Process Analysis	Multivariate Constrained Optimization and Sensitivity Analysis utility
Extras	Scripting System and Plugin Interface

The unit operations that come with DWSIM installation include a mixer, splitter, separator, pump, compressor, expander, heater, cooler, valve, pipe segment, shortcut column, heat exchanger, various reactors, a component separator, orifice plate, distillation/absorption columns, solids separators and a cake filter. Also, it is possible to create Excel, script and flowsheet based unit operations. (Medeiros et al. 2015c)

2.4.2 Commercial process simulators

A review article by van Baten and Pons (2014) gives an overview of the impact of the CAPE-OPEN standard on the simulation environment market, and assesses its industrial relevance. The article includes a list of 12 commercial process simulation environments that implement support for CAPE-OPEN, some of which are presented in Table 2.10.

Table 2.10. CAPE-OPEN compliant commercial PMEs (van Baten and Pons 2014)

Tool	Developed by	Remarks
AspenPlus	AspenTech	AspenPlus was one of the first CAPE-OPEN socket implementations available. In the early years this has been the de-facto standard for interoperability testing.
Aspen HYSYS	AspenTech	Hyprotech developed HYSYS before being purchased by AspenTech, and along with AspenPlus, HYSYS was one of the first CAPE-OPEN sockets around. The HYSYS product later on branched in Aspen HYSYS and Honeywell UniSim Design.
ChemCAD	Chemstations	The support for CAPE-OPEN in ChemCAD is currently (2014) limited to thermodynamics.
gPROMS	PSE	gPROMS implements a socket to use CAPE-OPEN thermodynamics for equation based problem set-ups. The gPROMS authors have been involved in CAPE-OPEN from the start.
ProSimPlus	ProSim	ProSimPlus is one of the few simulators that currently support CAPE-OPEN information streams in addition to material streams. The definition of information streams was completed relatively late in the unit operation standard specification.
UniSim Design	Honeywell	See the remarks for Aspen HYSYS.

According to van Baten and Pons, their list of compliant simulation environments is not exhaustive, and the same applies to the 11 more advanced CAPE-OPEN compliant unit operations and the 12 compliant thermodynamic servers listed in their review. Additionally, their review includes a general introduction to CAPE-OPEN, and presents results produced with the help of CAPE-OPEN standards.

3 DESIGN IN HSC CHEMISTRY SIM

This chapter begins with general information about the prerequisites for developing a unit operation model that would work in HSC Chemistry Sim. Relevant interfaces and classes are introduced. Additionally, select code examples are provided in order to illustrate the structure of a typical HSC Chemistry Sim unit operation. The latter half of the chapter contains information on the main development tools and programming languages used in the development of HSC Chemistry Sim, as well as a brief description of class library assemblies.

3.1 Requirements

There are fewer requirements for developing unit operation models for Sim than there are for developing similar unit operation models for CAPE-OPEN compliant PMEs, since there is no middleware such as COM involved, nor is there a need to adhere to any standards. However, in either case, it requires accurate knowledge about the unit operation to be modelled and adequate programming skills. In addition, access is needed to a computer that has a valid copy of HSC Chemistry installed, as the functionality for a unit operation depends on certain assemblies (.dll) that come with HSC Chemistry.

Some documentation is also needed. The HSC documentation set was amended by the author, see the 25-page document called Sim DLL Units Manual included as Appendix B. In cases where the developer is already familiar with Visual Basic and the .NET framework, it might in fact be comprehensive enough to allow him or her to start creating unit operations for Sim. However, subsequent to the writing of the manual, the HSC development team made changes to Sim, thus enforcing a revision to the manual.

It is most likely that development work will be done using Microsoft Visual Studio, although other alternatives exist. Whatever the choice of development tool, proper familiarity with the .NET framework is most beneficial. A thorough and well-organized book like (Boehm 2013) can be recommended to anyone new to the topic.

To make the initial steps of the unit operation model development work as easy to take as possible, the HSC development team has created a template file for unit operation models, see Section 3.1.2 for details. As such, the use of templates can be regarded as one of the better ways to remove unnecessary implementation details from the developer's path. They also help focus on the modelling work by hiding the plumbing code around the unit operation model.

The point of view expressed in Sections 3.1.1 and 3.1.2 is intentionally rather vague, due to the non-public nature of the source code for HSC Chemistry Sim, and as mentioned above, to avoid the introduction of distracting technical details. However, the selected approach should still enable the reader to form an idea of the many differences and similarities between HSC Chemistry Sim and CAPE-OPEN compliant simulation environments.

3.1.1 Sim unit operations

The basic ideas in HSC Chemistry Sim do not deviate much from the ideas present in CAPE-OPEN PMEs. There are the quintessential units and streams, and the accompanying services provided by the simulator. Figure 3.1 shows a schematic diagram for a Sim Unit Operation.

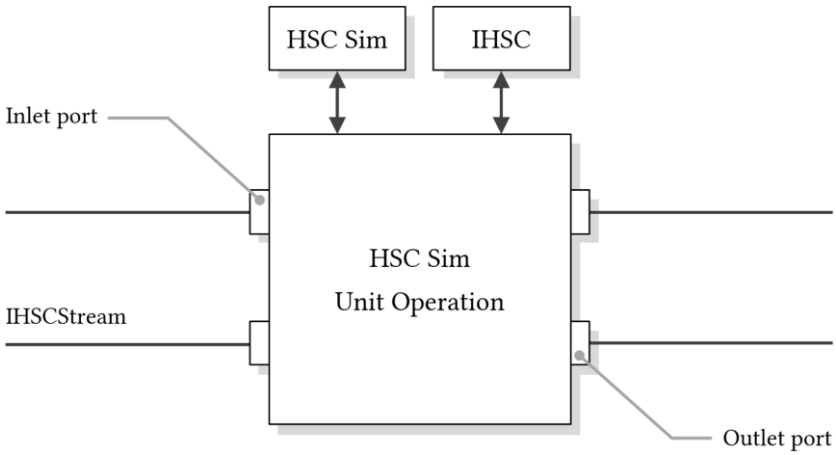


Figure 3.1 Schematic diagram for HSC Sim Unit Operation.

The feed and product, or input and output, streams of a unit operation are defined by the HSC8Stream class that implements an interface called IHSC8Stream, marked IHSCStream in Figure 3.1. The IHSCStream interface contains methods for handling the stream content. The streams themselves carry all the information necessary to define a material stream, and as such are the Sim equivalent to the CAPE-OPEN Material Objects. A well-defined stream includes information on composition, the phases that are present and the quantities of all constituent compounds, which are expressed in metric tons or tons/hour.

The concept of port is rather diminished in Sim as the streams are plugged into the unit operations in the source code through the use of Properties, which are a Visual Basic feature combining conventional Get and Set methods. The attachment of streams to a unit operation is exemplified in Figure 3.4 on page 54. Whether a stream is an input or an output stream is defined by the use of the custom Visual Basic attributes StreamIn and StreamOut.

Unit operations may or may not have all of the property calculations, equations and data they need on board the unit. In either case, the unit can also use the services provided in the IHSC interface. The calculation routines in the unit operation receive an instance of an HSC object passed in as a parameter, cast as IHSC. The IHSC interface has a role similar to the role of CAPE-OPEN Property Packages, as it contains calculation routines often needed in process simulations, such as methods for calculating enthalpies and densities.

Between the unit operation and the user interface is the HSC Sim engine, which is marked HSC Sim in Figure 3.1. This engine is the HSC Sim equivalent of a CAPE-OPEN PME. During a

simulation, the HSC engine takes care of transmitting information between different tiers of the simulation application that are not directly connected to each other, as shown in Fig. 3.2.

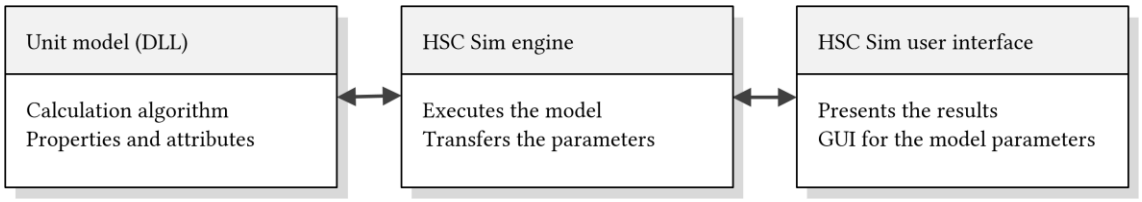


Figure 3.2 Overview of the program structure (adapted from Remes and Kentala 2014).

Fig. 3.3 presents the idea of the bidirectional relay of information between a unit operation and the user interface. Some of the user interface elements are created dynamically based on the type of unit operation, while some of the functionality is unchanged across all unit operations.

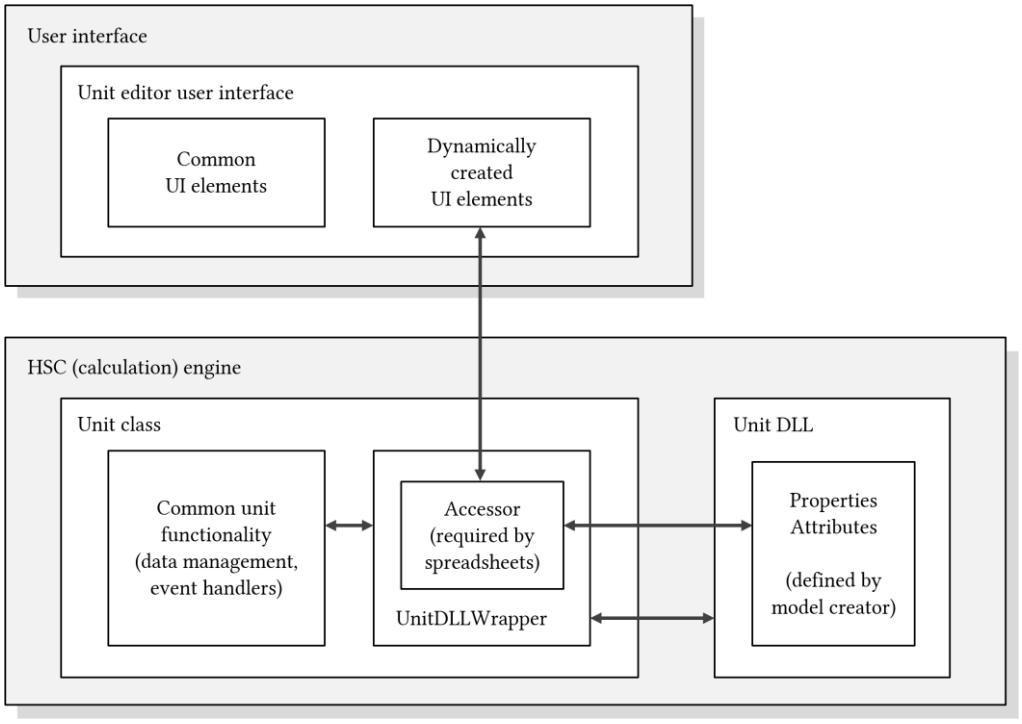


Figure 3.3 Simulation environment components (adapted from Kentala 2015).

As already shown in Figures 3.1, 3.2 and 3.3, an essential part of the unit operation models are the various parameters and attributes. The HSC Sim engine allows the user to access various unit parameters and attributes with the help of the panels and dialog boxes present in the user

interface. Sim unit operations have various parameters that are divided into four categories: model parameters, internal states, runtime variables and model fit parameters. For example, a typical model parameter inherent to a model of a tank would be its target fluid level. For a more comprehensive description, please see pages 8–18 in the User Manual in Appendix B.

It is worth noting the possibility of sending messages from unit operations to the HSC Sim user interface, as described in the User Manual on pages 19 and 23. These messages fall into four categories: log entries, info messages, warnings and error messages. The sent messages are shown in real time in the Log viewer, which is part of the user interface.

3.1.2 Implementation schema

The HSC unit operation development utilizes a code template that comes with HSC Chemistry installation. It is written in Visual Basic, and as such forms a starting point for development. A shortened version is shown in Figure 3.4; the full code listing is presented in Appendix C.

```
Imports System.AddIn
Imports System.AddIn.Pipeline
Imports Outotec.HSC.Addins

<AddIn("Name")>
<QualificationData("Static", "True")>
<QualificationData("TypeCode", "MU-100-10")>
<QualificationData("Version", "1.0")>
<QualificationData("Description", "Description")>
Public Class _Template
    Inherits UnitDLL

    'Inputs
    <StreamIn("Input", 1)>
    Public Property Input As IHSC8Stream

    'Outputs
    <StreamOut("Output", 1)>
    Public Property Output As IHSC8Stream

    Protected Overrides Sub CalcStatic(ByVal HSC As IHSC)
    ' ...
    End Sub

    Protected Overrides Sub CalcDynamic(ByVal HSC As IHSC, ByVal timestepsseconds As Integer)
    ' ...
    End Sub

End Class
```

Figure 3.4 Shortened version of the HSC Sim unit operation template.

The input and output streams and the unit operation parameters are defined as Public Properties, as using any other access modifier would likely render the Properties useless. The source code also contains Attributes, which are <marked like this> in Visual Basic. The Attributes are tags that provide additional information to describe programming elements. The Attributes in the template are used to specify information like the name and version of a unit. All the streams are tagged with Attributes to determine whether they are input or output streams. Appendix B contains more examples of the use of Attributes, see pages 14 and 16.

The unit operation class must inherit a base class called *UnitDLL* (Kentala 2016). Inheritance enables effective code reuse. The relationship between the classes is conceptually very similar to the one presented in Figure 2.22.

The Imports statements in Figure 3.4 and the AddIn attribute below them manifest the use of the .NET Framework add-in model. It provides a programming model that can be used to develop add-ins which can be activated in their host applications, and is remotely reminiscent of the COM techniques used in CAPE-OPEN. Only this time, everything is expressed with a mere three lines of code. The Imports statements simplify code typing as they remove the need to fully qualify class names. Their use requires that appropriate references to relevant class libraries be made, see pages 6–7, 11 in Appendix B for further information.

Note the presence of the two interfaces, IHSC8Stream and IHSC, in Figure 3.4. The latter is passed into the calculation routine as a parameter. These are the only proprietary interfaces that a developer needs. Appendix D contains the source code for a sample unit operation called the Perfect Mixer, which demonstrates the use of selected IHSC8Stream methods.

Once the unit operation class is compiled, it is deployed in the form of a class library assembly (.dll). Currently, the deployment practice requires that the assembly be copied to a subfolder under (HSC)\Sim\Addins81\Addins (Kentala 2016). Sim makes note of the newly added assembly and adds the unit operations that it finds in it to the appropriate menus within the GUI. Appendix B (page 21) contains further information.

Lastly, in order to allow for a minor comparison between approaches and to give an idea of the amount of work involved, snippets from the DWSIM source code are presented in Figures 3.5, 3.6 and 3.7. The source is available from various websites, as described in Section 2.4.1.

```
Imports System.Runtime.Serialization.Formatters.Binary
Imports DWSIM.DWSIM.SimulationObjects.PropertyPackages
Imports System.Runtime.InteropServices.Marshal
Imports Microsoft.MSDN.Samples.GraphicObjects
Imports DWSIM.DWSIM.Flowsheet.FlowsheetSolver
Imports Microsoft.Scripting.Hosting
Imports System.Linq
...

<System.Serializable()>
<ComVisible(True)>
Public MustInherit Class SimulationObjects_BaseClass

    Implements ICloneable, IDisposable, XMLSerializer.Interfaces.ICustomXMLSerialization

    ' member variables, methods
    ' approximately 3400 lines of code
    ' ...

End Class
```

Figure 3.5 DWSIM *SimulationObjects_BaseClass*.

At the bottom – or top, if preferred – of the inheritance tree lies a rather extensively coded class named *SimulationObjects_BaseClass*. Every other object to be used in the simulation has to inherit this class. As can be seen, it is made to work under COM and uses the .NET frame-

work serialization techniques; here the class is serialized to an XML file. Another commonly used alternative is serialization to a JSON file. The *SimulationObjects_UnitOpBaseClass* class presented in Figure 3.6 inherits the *SimulationObjects_BaseClass*, and implements the CAPE-OPEN interfaces, including the persistence and error interfaces.

```
<System.Serializable()>
<ComVisible(True)>
Public MustInherit Class SimulationObjects_UnitOpBaseClass
    Inherits SimulationObjects_BaseClass

    'CAPE-OPEN Unit Operation Support
    Implements ICapeIdentification, ICapeUnit, ICapeUtilities, ICapeUnitReport

    'CAPE-OPEN Persistence Interface
    Implements IPersistStreamInit

    'CAPE-OPEN Error Interfaces
    Implements ECapeUser, ECapeUnknown, ECapeRoot

    Private _pp As DWSIM.SimulationObjects.PropertyPackages.PropertyPackage

    ' member variables, methods, data
    ' approximately 800 lines of code
    ' ...

End Class
```

Figure 3.6 DWSIM *SimulationObjects_UnitOpBaseClass*.

The DWSIM base classes are counterparts to the HSC *UnitDLL*, as the *CustomUO* class, part of which is shown in Figure 3.7, inherits the two classes described above.

```
Imports DWSIM.DWSIM.Flowsheet.FlowsheetSolver
Imports System.Runtime.InteropServices
Imports CapeOpen
...

Namespace DWSIM.SimulationObjects.UnitOps

   <Guid(CustomUO.ClassId)> <System.Serializable()> <ComVisible(True)>
   Public Class CustomUO
      Inherits SimulationObjects_UnitOpBaseClass

      ' member variables, methods, ClassId
      ' approximately 1000 lines of code
      ' ...

    End Class

End Namespace
```

Figure 3.7 DWSIM *CustomUO* class, which enables IronPython or IronRuby scripting.

Interestingly, a DWSIM user does not have to work with the DWSIM classes in this form at all. Substantial plumbing enables flexibility and provides for easier use, since the functionality of custom unit operations is programmed into the models in the IronPython or IronRuby languages. Custom unit operations come with dedicated script editor dialog boxes.

3.2 Development and deployment

The agile development work of a complex software suite carried out by a team of developers scattered across Europe involves the use of multiple development tools. These include issue tracking products, distributed version control tools, integrated development environments (IDEs) and others. In essence, the most important of these tools is the IDE used for writing the actual software code. This section introduces the Microsoft Visual Studio IDE briefly, and discusses the Visual Basic (VB) programming language and the affiliated .NET Framework, since all of them are used at Outotec in their software development work.

3.2.1 Visual Studio Integrated Development Environment

Visual Studio is a suite of products for creating software, and includes tools for designing user interfaces, code writing, testing, debugging, analysing code quality and gauging performance, deploying products, and gathering telemetry on usage. In the 2012 version of Visual Studio, the supported programming languages are Visual Basic, C#, Visual C++ and F#, and the 2015 version, the latest version at the time of writing, also supports development with JavaScript and can be expanded to support other languages like Python. (Microsoft 2016b; Boehm 2013)

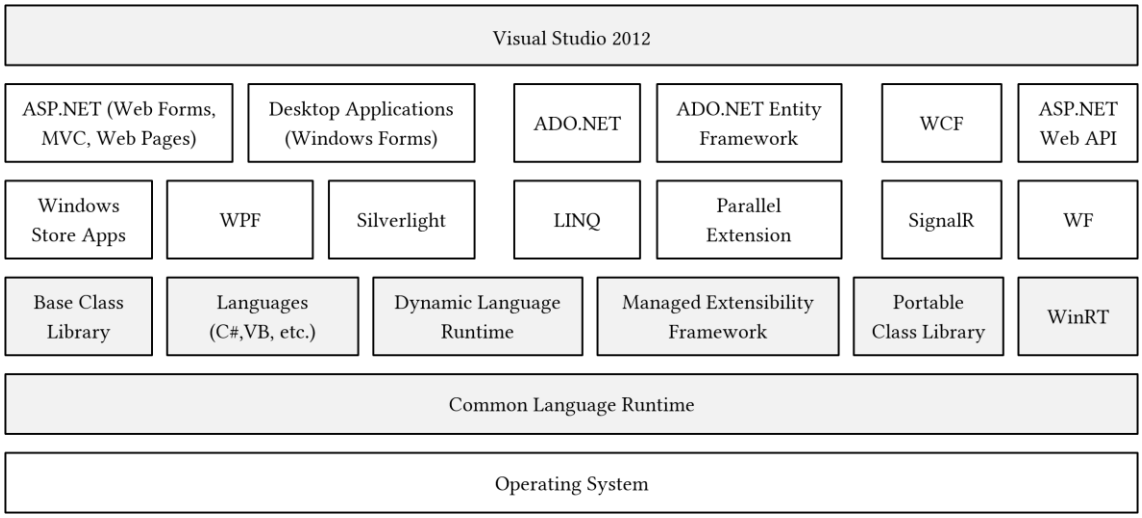


Figure 3.8 .NET Framework 4.5 architecture (adapted from Chauhan 2014).

Visual Studio is available in different editions that cater for the needs of various user groups. There is a free edition for students and hobbyists and non-free editions with extra features for individuals and teams. The .NET Framework Class Libraries, which include the Base Class Library and most of the items in the white boxes in rows two and three of Figure 3.8, provide a common set of services that applications written in a .NET language can use to run on various operating systems and hardware platforms. Visual Studio can be used to build a wide variety of Windows and web, mobile, and Microsoft Office-based application. (Boehm 2013)

3.2.2 Object-oriented Visual Basic and the .NET framework

Visual Basic has gained popularity due to its increased productivity in comparison to other programming languages when building business applications (Sheldon et al. 2013). Visual Basic and C# have been designed for rapid application development (Boehm 2013).

Microsoft released .NET in 2002 as an update to COM and other related technologies. Visual Basic had existed before that, but went through a major overhaul. The language changed from being object-based to object-oriented and it can be now regarded as a fully-fledged object-oriented, event-driven programming language. (Sheldon et al. 2013) Although it has the word Basic in its name, it has virtually nothing to do with any of the BASIC languages of the past.

The .NET Framework Class Library (Figure 3.8) consists of pre-written code called classes, which offer many of the functions that a developer needs for developing .NET applications. As an example, the Windows Forms classes can be used when developing Windows Forms (desktop) applications. There are classes that enable one to work with databases, manage security, access files, and perform a plethora of other tasks. (Boehm 2013)

The Common Language Runtime (CLR) implements a virtual machine that interprets byte code that is a hardware-independent, intermediate format between a high-level language such as Visual Basic and the machine code executed by the central processing unit. The byte code for the .NET framework is a language called common intermediate language (IL or CIL for short), also sometimes called Microsoft Intermediate language (MSIL) (CO-LaN 2006). All of the .NET languages compile to CIL. (Boehm 2013)

The CLR also provides the Common Type System (CTS) that ensures that all .NET applications use the same basic data types no matter what programming languages are used to develop the applications. (Boehm 2013) By sharing the type definitions, language interoperability becomes quite simple, as there is no need for data conversions. In addition, as all languages share a representation of their code in the intermediate language, all function definitions are present in a compatible form as well. Hence, calling functions across languages or application boundaries is also a fairly simple issue and is completely taken care of by the CLR. In a sense, the same algorithm written in different programming languages, such as C#, VB or C++, can be thought of as different views of the same problem, since the underlying IL is nearly identical. (CO-LaN 2006) Because .NET applications are managed by the CLR, they are called managed applications (Boehm 2013).

Consequently, the language in which software components have originally been written becomes completely transparent to developers. Instead of just calling methods on behalf of a foreign component regardless of the languages involved in the implementation, they can even inherit from code constructed in other languages, including method overloading, and common issues such as serialization or exception handling are handled by the CLR in a completely transparent fashion. (CO-LaN 2006)

One of the advantages of executing an application in a virtual environment like the CLR, instead of allowing the application to access hardware directly, is the fact that the CLR can guard the memory usage of the application and perform a so-called garbage collection that frees objects no longer referenced by the application. As a result, stability and safety are

improved and there is reduced effort on the part of the developer to spend time on reference counting and other tasks inherent to developing non-managed software. (CO-LaN 2006)

Figure 3.9 shows how a Visual Basic application is compiled and run. Development starts by using Visual Studio to create a project, which is made up of source files that contain Visual Basic statements, see step 1 in Figure 3.9. A project may also contain other types of files, such as sound, image, or text files. (Boehm 2013)

After the Visual Basic code has been entered for a project, the Visual Basic compiler is used to build (or compile) the Visual Basic source code into an intermediate language, see step 2 in Figure 3.9. At this point, the intermediate language is stored on disk in a file called an assembly, which is an executable file that has an .exe or .dll extension. In addition to the IL, the assembly may include references to other files that contain classes that the application needs in order to run correctly. The assembly can then be run on any computer that has the CLR installed on it. At runtime, the CLR converts the intermediate language to native code that can be run by the Windows OS; see steps 3 and 4 in Figure 3.9. (Boehm 2013)

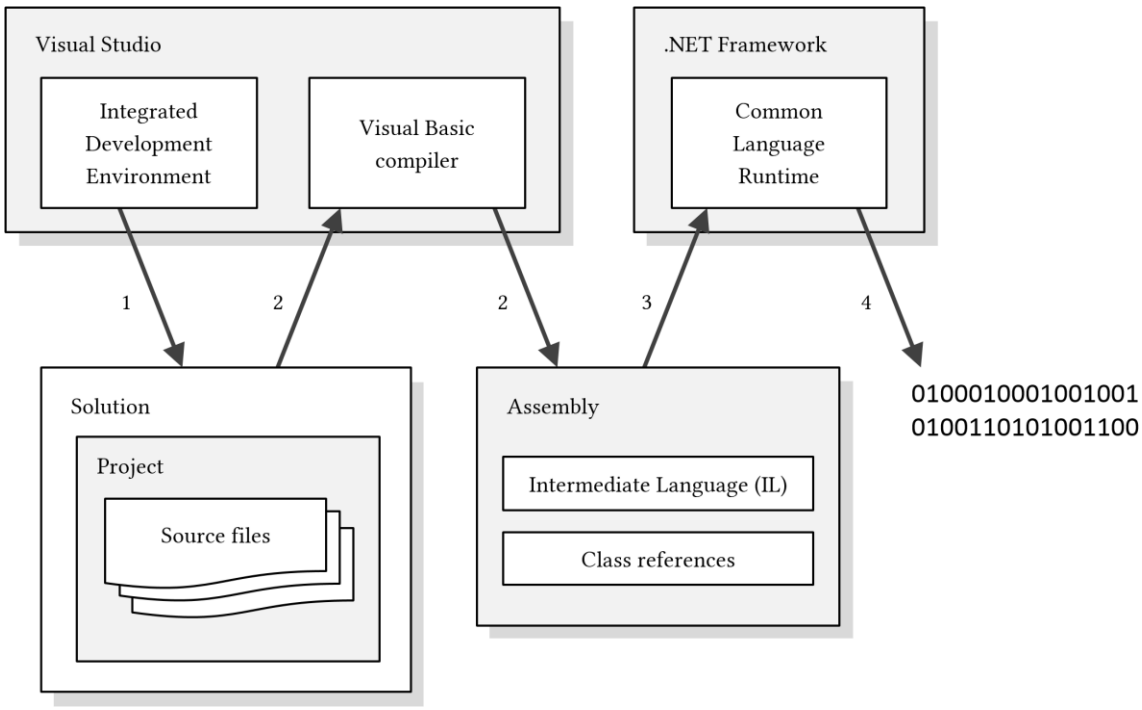


Figure 3.9 How a Visual Basic application is compiled and run (adapted from Boehm 2013).

Furthermore, when the byte code (the IL) resulting from the compilation of an application is not interpreted directly as described above, but run through a Just-In-Time (JIT) compiler (within the CLR), it is executed when the application is started. In fact, the speeding up of executing compiled code versus interpreting byte code by far outweighs the overhead of the just-in-time compilation step itself. In conclusion, the performance of .NET applications is approximately the same as for applications developed using unmanaged code. (CO-Lan 2006)

3.2.3 Dynamic-link libraries

A dynamic-link library (DLL) is a collection of code and data that can be called by more than one application at the same time. As opposed to a static link that remains constant throughout the execution of a program, a dynamic link is created when needed. (Microsoft 2016c)

Consider the two Windows Forms projects in Figure 3.10. Both projects utilize InputValidators that are instances of a class called Validator used for validating data. For example, if the NewCustomerForm had a field for the customer's age, the user input would be passed to the InputValidator in Project A to check that the input consists of only numbers and it could return an error if the age was not an integer. The same validation routine could be used similarly in conjunction with the NewEmployeeForm in Project B.

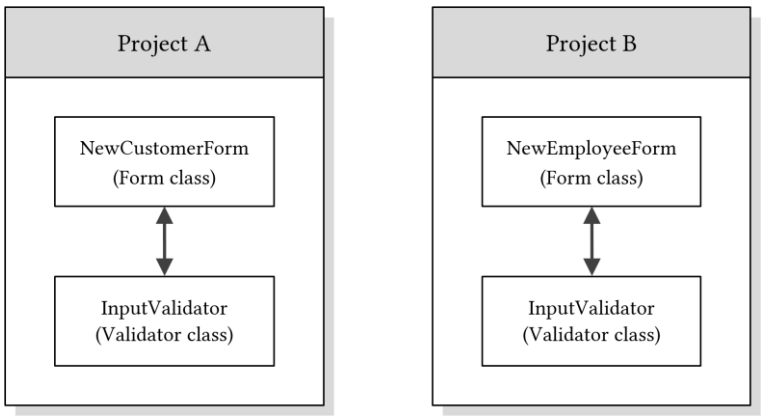


Figure 3.10 Two separate projects that use the Validator class for input validation (adapted from Boehm 2013).

Alternatively, the InputValidator class can be included in a class library project. A class library project would compile to a DLL file that can be referred to from any number of projects needing input validation services, see Figure 3.11.

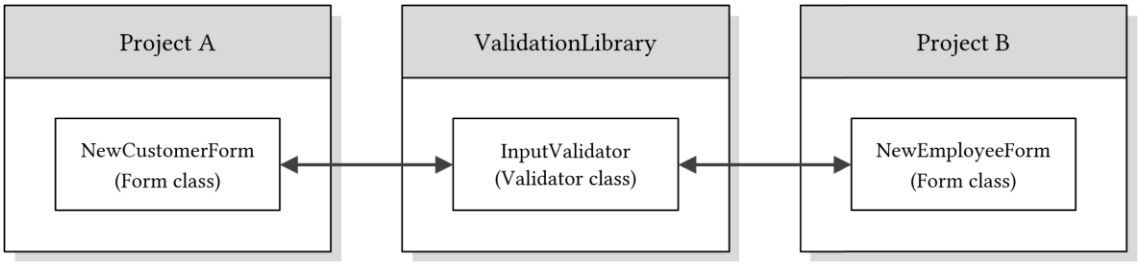


Figure 3.11 Two separate projects that access the Validator class via a class library (adapted from Boehm 2013).

The benefits of this kind of arrangement are that the size of each project using shared DLLs is reduced. The reason for this is that each project includes only a reference to the DLL rather than the code that is in the DLL. The maintenance is simplified, too. If changes have to be made to a class in a class library, they can be done without the need to change anything in the applications that use the library. Another benefit of DLLs is that they allow a developer to create reusable code. Well-designed classes can be reused in later projects, because of their functionality or as base classes for new classes. (Boehm 2013)

In general, dynamic-link libraries constitute a broad topic. Technically, COM-based DLLs that contain unmanaged code are quite different from their .NET-managed code counterparts. In the .NET framework, compiled class library files, such as the ValidationLibrary class library depicted above, are called assemblies as they may contain several possibly quite diverse elements, instead of just intermediate language code, as shown in Figure 3.12. Class library assemblies have the extension .dll in the file names.

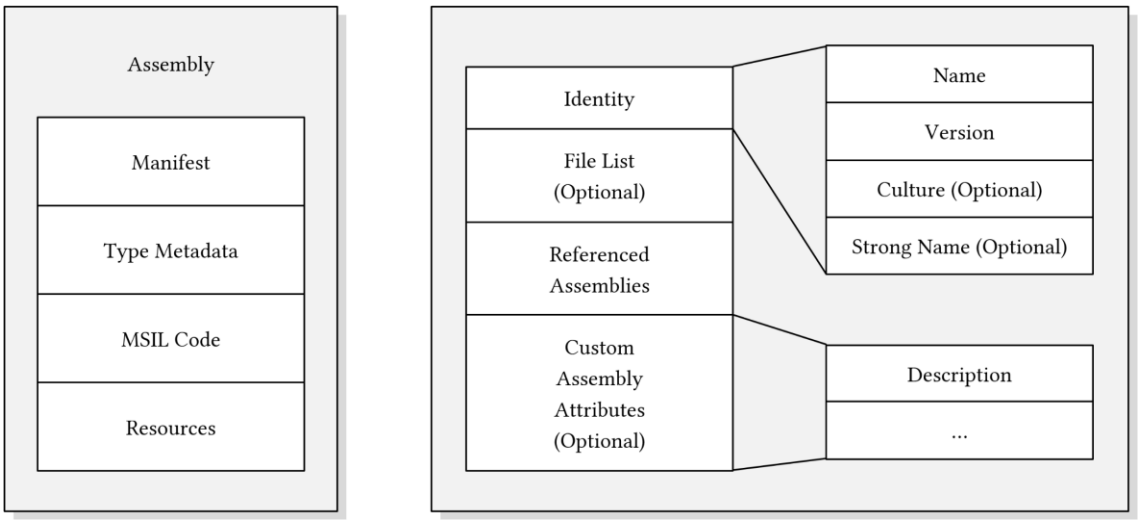


Figure 3.12 Logical structure of an assembly (on the left) and logical structure of an assembly manifest (on the right) (adapted from Sheldon et al. 2013).

Assemblies in .NET usually consist of the assembly manifest, metadata describing types such as the various classes, interfaces and enumerations contained in the assembly, the (Microsoft) intermediate language code that implements the types, and a set of possible resources. Everything else apart from the manifest is optional. The elements that make up an assembly can also be combined in different ways. A single .dll file can contain an entire assembly or the assembly content can be spread across multiple files, as shown in Figure 3.13 overleaf. (Sheldon et al. 2013; CO-LaN 2006)

The manifest has an item called assembly identity, which is shown on the right-hand side of Figure 3.12. It includes a name, which is the same as the filename of the assembly, a version number, and an optional culture for defining the country or language for which the assembly

is targeted. The identity can also contain an optional strong name. A strong name is not actually a name, but a security measure that involves the inclusion of a public key to an assembly. The public key is generated by the author of the assembly against his or her private key. In addition to the assembly identity, a manifest can also include a list of all the files that make up the assembly, and a list of referenced assemblies if the assembly is dependent on other assemblies. (Sheldon et al. 2013; CO-LaN 2006)

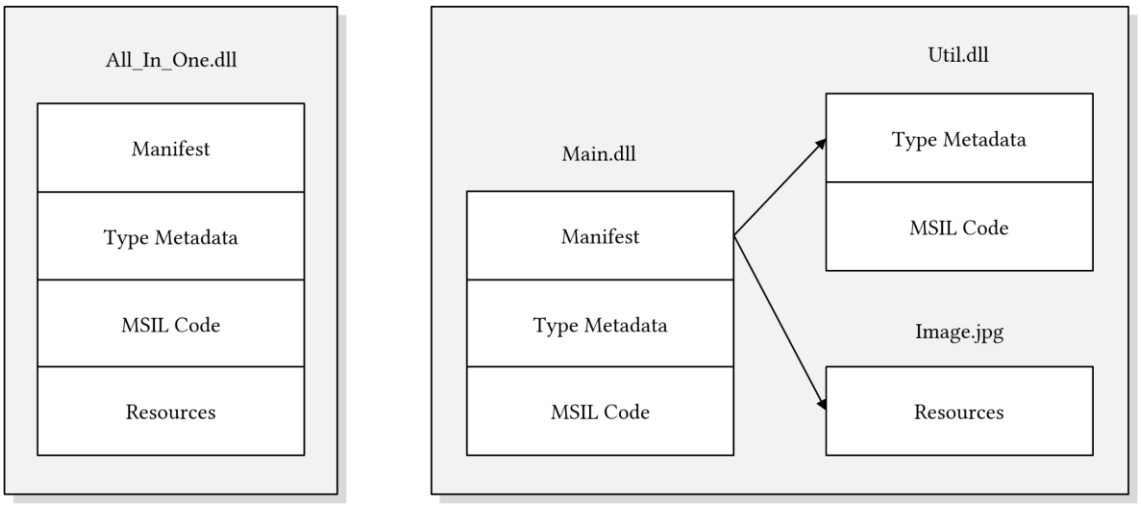


Figure 3.13 Assembly contents (adapted from CO-LaN 2006).

Assemblies used by a particular application are usually located in a folder known to the application. With the metadata about the types and resources contained in the assembly, it is possible to inspect the contents of an assembly by a technology called reflection. On the other hand, assemblies with strong names, those with a public key, can be placed in a central area called the global assembly cache (GAC). Assemblies are added and removed from the GAC by using the utility tool gacutil.exe that is shipped with .NET. The use of a GAC removes the need to have different copies of a single assembly at various locations on the computer and permits different versions of an assembly with the same name to coexist in one folder, which is not possible in an ordinary folder. (Sheldon et al. 2013; CO-LaN 2006)

As explained in Section 3.1.2, the file containing a model for a unit operation to be used in HSC Chemistry Sim is a class library file. This class library, or assembly, file is very much like the assemblies described in this section.

COM and .NET interoperability

When a complex application programming interface (API) such as CAPE-OPEN is updated from one version of middleware to another, various implementation issues arise. There are some areas where .NET implementations must be aware of the peculiarities of COM to interoperate properly. Obvious issues include data types/values, exceptions and persistence where .NET has made significant departures from COM. (CO-LaN 2006)

Microsoft has put substantial effort into mechanisms that permit the interoperability of code provided for the .NET environment (for example the managed code DLLs discussed above) with unmanaged code, such as the classic DLLs that were developed using C++ or Fortran before the .NET era or various COM components that were developed to permit interoperability of applications. (CO-LaN 2006)

When a COM component is to be used in a .NET application, an intermediate piece of software is needed to translate between the .NET set of types and the types that are defined in COM, see Table 3.1, and to expose the interfaces implemented by the COM components in a .NET-compatible way. This piece of software is termed a runtime callable wrapper (RCW), since the CLR can natively call the wrapper around the COM component, see Fig. 3.14. (CO-LaN 2006)

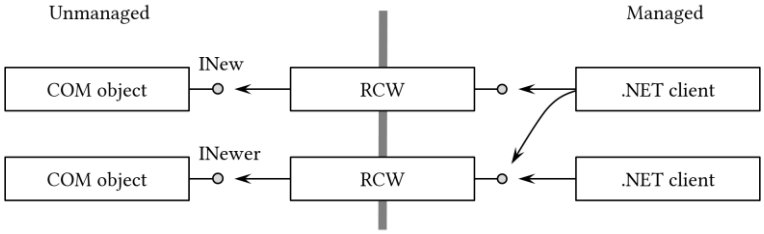


Figure 3.14 Accessing COM objects through the runtime callable wrapper (adapted from Microsoft 2016d).

This intermediate piece of software can be generated in a fully automated fashion from a type library (.tlb), which is usually present for COM components meant to be used by other applications. The .NET framework contains the tool tlbimp.exe, which is given a type library and generates an assembly containing the RCW from it. (CO-LaN 2006)

The RCW basically translates the COM data types into their equivalent .NET data types. Table 3.1 provides a sample list of the CAPE-OPEN data types, their COM equivalents and the data type that the interop marshalling within the RCW would provide to .NET. (CO-LaN 2006)

Table 3.1. Comparison of CAPE-OPEN, COM and .NET data types (Barrett et al. 2005)

CAPE-OPEN data type	Description	COM data type	.NET data type
CapeDouble	Double	Double	Double
⋮			
CapeString	String or char[]	BSTR	String
CapeDate	String date	VARIANT_DATE	String
CapeInterface	CO interface	LPDISPATCH	Object

When implementing a COM component using the .NET framework, the .NET framework reduces the effort spent by the developer by automatically generating an intermediate piece of software that deals with the interaction of the .NET component code and the COM runtime system. This generated wrapper is called a COM callable wrapper (CCW) and implements COM interfaces based on the properties of the underlying code developed using .NET (see Figure 3.15). The main contribution of the CCW is translating types between the common type system and the COM type definitions, handling reference counting, type casting, and implementing the essential interfaces defined in COM. (CO-LaN 2006)

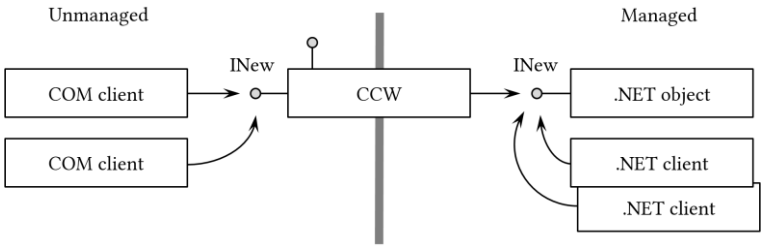


Figure 3.15 Accessing .NET objects through COM callable wrapper (adapted from Microsoft 2016e).

In both cases, the intermediate piece of software (RCW, CCW) that is required to translate between .NET and COM is called an interop assembly. A particular type of interop assembly is the primary interop assembly (PIA), which is signed by the component publisher and marked as belonging to the COM component. The code snippet in Figure 3.16 exemplifies how to launch an instance of Microsoft Excel – through the PIAs delivered by Microsoft – then load a workbook from the test.xls file and activate it. (CO-LaN 2006)

```
Microsoft.Office.Interop.Excel.Application excel = null;
Microsoft.Office.Interop.Excel.Workbook wb = null;
object m = Type.Missing;

excel = new Microsoft.Office.Interop.Excel.Application();
wb = excel.Workbooks.Open("c:\\test.xls", m, m, m, m, m, m, m, m, m, m, m, m, m, m);
excel.Visible = true;
wb.Activate();
```

Figure 3.16 Launching Microsoft Excel from an application (adapted from CO-LaN 2006).

CAPE-OPEN error handling deviates from the COM *GetErrorInfo* API. As a result, .NET-based objects cannot use .NET exceptions alone to handle errors, but have to support the Error Common Interface, and have to implement all the error interfaces (CO-LaN 2003g) that they are able to raise. Also, persistence requires the use of an interface called *IPersistStreamInit*, the use of suitable wrapper classes or the use of .NET serialization procedures, see (CO-LaN 2006).

4 RESULTS AND DISCUSSION

All the results presented herein are essentially qualitative in nature, as there are no suitable quantitative metrics to apply. Aspects regarding software engineering were at the forefront in this study, yet at the core the development of process simulation components is always an interdisciplinary challenge.

In order to create a CAPE-OPEN compliant model of a real world unit operation, the modeller has to acquire a working understanding of the CAPE-OPEN interface set. Also, it is imperative to have a sufficiently high level of expertise in programming and software design. The prerequisites for model development are summarized in Figure 4.1.

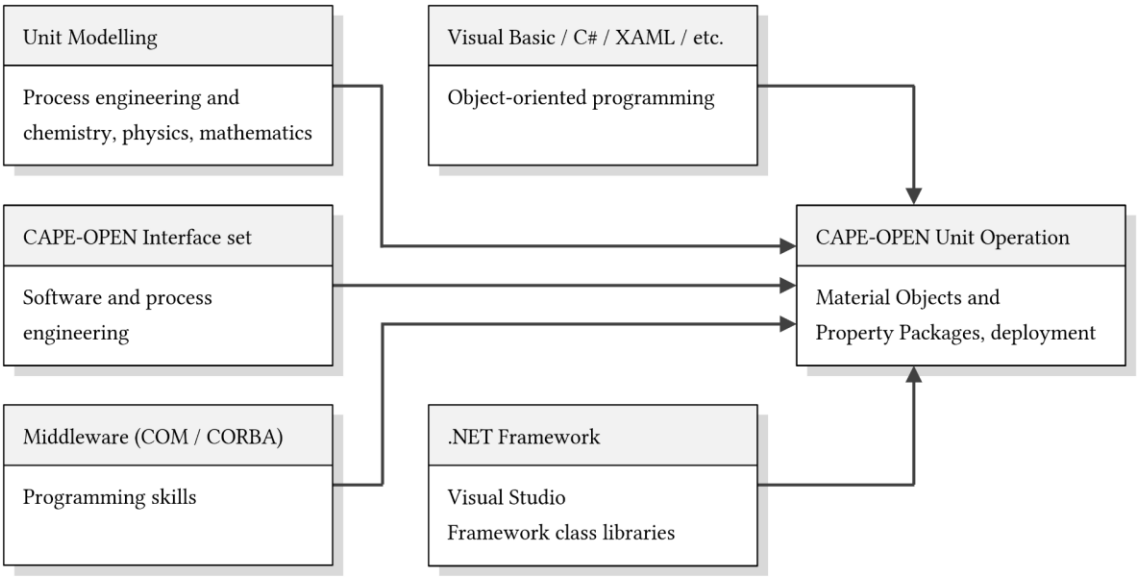


Figure 4.1 Overview of the skill requirements for creating CAPE-OPEN Unit Operations.

The CAPE-OPEN interface set has been in use for more than a decade now, and its position is well established. Numerous examples presented in this thesis demonstrate that the CAPE-OPEN interfaces are can be adopted. They enable the development of exchangeable process modelling components – even standalone reactors that come with built-in optimization routines – and make it possible to create a myriad of interesting applications that extend the scope of the usability of the interfaces. Extensions, add-ins and prototyping tools have gradually brought added value to the CAPE-OPEN interfaces.

However, the implementation of process equipment models in the CAPE-OPEN framework sets a considerable threshold. It does not suffice to simply create a unit operation with the right interfaces and collections of ports and parameters, but other objects like the Material Objects with their own interfaces have to be implemented, as unit operations are rendered useless without Material Objects.

The central role of Material Objects is made evident in other ways, too. Material Objects do not only contain stream compositions and values of state variables, but are also passed to the Property Packages or similar objects for property and thermodynamic calculations. Figure 4.2 shows a screenshot from DWSIM that illustrates the information carried in Material Objects.

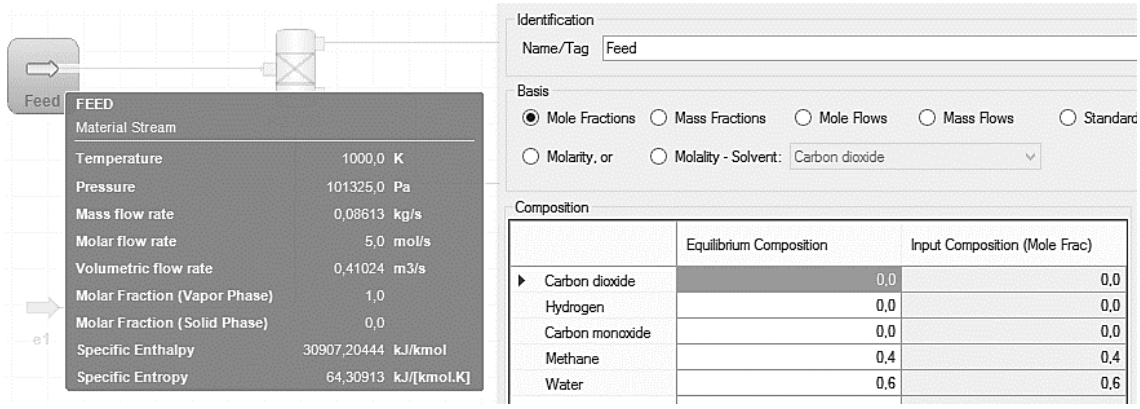


Figure 4.2 The values of state variables and properties, on the left, for a material stream, whose composition is shown on the right.

In fact, the Material Object represents the biggest difference between HSC Chemistry Sim and CAPE-OPEN. A CAPE-OPEN Material Object is a relatively complex object, whereas the HSC material stream object is fairly straightforward with scarce responsibilities. Conceptually, they are one and the same, but at the implementation level the only factor they seem to share is the inclusion of phase, composition and flow rate information.

Technically or, more precisely, mathematically, a unit operation in HSC Chemistry Sim could be almost identical to its CAPE-OPEN counterpart, if it were not for the differences in the implementation of thermodynamic and property calculations. HSC Sim unit utilizes the methods defined in the IHSC interface to manipulate the stream contents, while a typical CAPE-OPEN unit operation would request a Material Object to set itself to a state of thermodynamic equilibrium by means of property packages. The ambient conditions may depend on the parameter values set by the unit operation.

As process simulation components are modelled with equations, there is an inherent need for methods for solving these equations, most often simultaneously. Sometimes, general-purpose mathematical model solver utilities are employed. Alternatively, calculation services can be provided by the simulation environment itself. Additionally, it is possible to have the

calculation routines on board a unit operation. The examples presented in the text support the idea that perhaps the smartest way to implement a solver would be to make it conform to the principles of modular design, and to make it as reusable as possible.

Additionally, to ensure thermodynamic consistency across an entire flowsheet model, the use of central facilities for performing thermodynamic and property calculations is required. In the CAPE-OPEN compliant simulation environments, this is typically the task of the aforementioned property packages. All that was concluded in the previous paragraph about solvers applies in general to property packages. Modularity and reusability facilitate future development, as well as the inherent troubleshooting.

The inherent complexity of the CAPE-OPEN standards has been discussed in various papers (Pons 2005; Lajmi et al. 2009; van Baten 2009), and can be inferred from the number of tools and utilities that have been devised over the years to help developers to create CAPE-OPEN compliant components. However, an assessment of the usefulness of some of these aids shows that time and technological changes have made them partially or completely obsolete.

One of the objectives of this thesis was to cover the deployment of the unit operation models to be used in HSC Chemistry Sim. Basically, the deployment involves the creation of a class library assembly, a DLL file, which is copied to a particular folder. Sim then adds the applicable contents of the class library into its selection of available unit operations. The deployment in COM-based applications was shown to be more complex.

Also, some elementary information on the Visual Basic programming language, the Visual Studio integrated development environment and the .NET Framework were presented as they form the core of the toolset used in the development of HSC Chemistry Sim and its unit operations. There is an underlying realization that when uniting a programming language designed for rapid application development with an extensive framework to support that development, the result is an increase in work efficiency. Additionally, the developer is freed to have more time to think about the design aspects, hence helping to improve the application.

With regard to research material, it seems that most of the information concerning CAPE-OPEN can be obtained almost exclusively from online sources, such as the CO-LaN website. The COCO website and the DWSIM documentation and source code are also regarded as useful sources of information, alongside scientific papers and lecture notes written by researchers from academia and the chemical process industry. The material concerning HSC Chemistry Sim was provided by the HSC Chemistry development team, and for the most part is not public. No previous studies are known to exist in the same field.

On the whole, the grand entity known as the CAPE-OPEN interface set was scrutinized and the findings were compressed into a thesis-sized literary work. In the process, a comparison of development approaches was made possible. Moreover, this thesis can serve as a basis for further discussion at Outotec concerning the future development of HSC Chemistry Sim.

5 CONCLUSIONS

As it stands, CAPE-OPEN technologies provide a well-tested and verified blueprint for several aspects of simulator design. Various alternative developmental paths based on the CAPE-OPEN ideas can be envisaged, see Table 5.1. Some of them may potentially improve the functionality, marketability or even usability of Outotec HSC Chemistry Sim. It should be noted that CAPE-OPEN-based implementations can be distributed without any CAPE-OPEN-related licence or intellectual property restrictions, see for example (van Baten and Pons 2014).

Table 5.1. Alternative measures

Measure
Take no action regarding CAPE-OPEN The easiest course of action, simply continue as if CAPE-OPEN did not exist.
Adapt some of the CAPE-OPEN practices and techniques Making minor changes to Sim by utilizing some parts of the CAPE-OPEN approach could help troubleshoot technical problems; the idea of more active stream objects is also worth considering.
Production of commercial CAPE-OPEN components For example, the HSC class with its calculation routines together with relevant databases could be converted into a CAPE-OPEN Property Package.
Partial implementation Making stream objects and unit operations adhere to the CAPE-OPEN standards would allow the use of third party components in HSC Chemistry Sim.
Complete adaptation of HSC Chemistry Sim to CAPE-OPEN This alternative would require a complete overhaul of most objects related to Sim, and would include major changes to Sim itself.

The inclusion of the first item in Table 5.1 might seem a little odd at first sight. As such, Outotec is not obliged to make any of their products CAPE-OPEN compliant. In a way, it is there for the sake of completeness, but also because there is a substantial code base for the present implementation of HSC Chemistry Sim, which has required time, money and other resources

to produce. Before ignoring this option and thus discarding any existing code, there must be valid, thought-out arguments in favour of at least one of the other options listed in Table 5.1.

Partial adoption of the ideas and solutions included in the CAPE-OPEN documentation set seems feasible. CAPE-OPEN projects have had participants from some of the biggest companies in the industry, along with researchers from several universities and research centres. They all shared their insights and experience to the advantage of CAPE-OPEN. As listed in Table 5.1, the benefits could include a decrease in time spent on troubleshooting and mending broken code. Also, the idea of using the material streams as containers for more data would separate wider sections of data, such as state variables, away from the user interface to the underlying data model. This would lead to a better conformity with the model-view-controller (MVC) architectural pattern. Additionally, object serialization would become more realizable.

Production of commercial CAPE-OPEN components is possible for anyone with a computer and enough know-how. The HSC development team has gained experience in developing applications for process engineering and educational purposes. It is only a matter of finances and interest if they would like to seek new ways to capitalize on the existing intellectual property.

Partial compliancy with CAPE-OPEN can be achieved with the use of wrappers, which is the anticipated way to start making CAPE-OPEN compliant components (CO-LaN 2000). However, the structural details of how a suitable wrapper ought to be designed is beyond the scope of this work. Nevertheless, a schematic diagram for illustrative purposes is shown in Fig. 5.1.

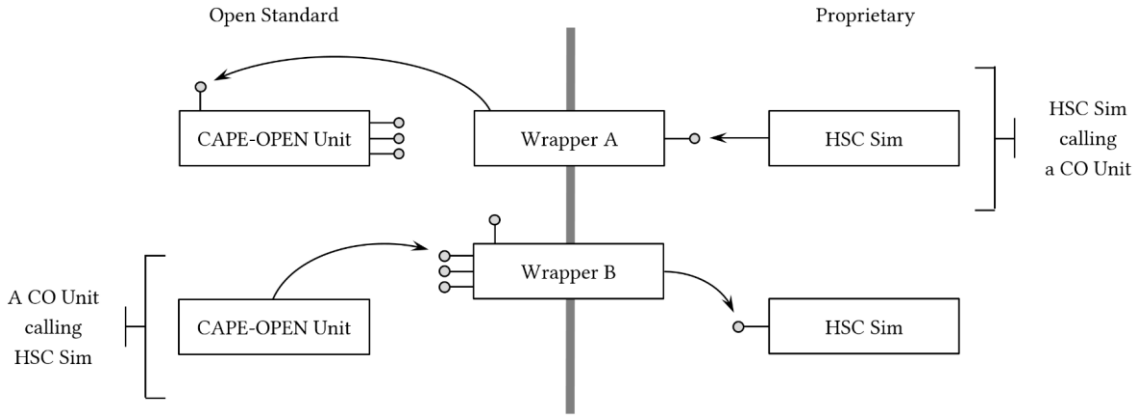


Figure 5.1 Application of wrappers (adapted from Microsoft 2016f).

In fact, it is most likely that only one wrapper per component is needed, as was demonstrated by Barrett and Yang (2005). A wrapper can also be a filter-like arrangement that makes sure that only the necessary functionality of a more diverse object is employed.

Complete adaptation of HSC Chemistry Sim to CAPE-OPEN, the last item in Table 5.1, is the most demanding and the most time-consuming option. On the positive side, the implementation of CAPE-OPEN interfaces results in immediate interoperability with well-

known, established simulation applications. This option would also provide the users with increased flexibility, as they would be able to select the most suitable software components from a wider selection to meet the needs of their particular problems. In addition, the use of CAPE-OPEN interfaces ensures consistency of simulation results across different process simulation environments.

Conformance to the CAPE-OPEN standard may also bring other, perhaps more immaterial benefits along. In fact, it is open to speculation whether the CAPE-OPEN emblem on HSC Chemistry Sim wrapping would arouse more interest in Sim in the process industry and academia, as CAPE-OPEN compliancy may be seen as a sign of reliability, even credibility. There would certainly be more publicity, if CO-LaN and its affiliates started listing HSC Chemistry Sim as a compliant product in their media, such as product catalogues, newsletters and websites. Who knows, maybe even the existing customer base and trade journals would greet the change with a favourable response.

On a related note, Appendix E presents the usability heuristics created by Nielsen (1995). These are general principles for interaction, and are called heuristics because they are broad rules of thumb and not exact, specific usability guidelines. CAPE-OPEN compliant DWSIM scores highly when evaluated in terms of these heuristics. Perhaps the CAPE-OPEN object hierarchy lends itself to better user experience and user interface design.

In a nutshell, the development of CAPE-OPEN components requires time, which in turn translates into money. The amount of time that the changes would take is hard to assess, and depends directly on the magnitude of the changes to implement. However, the invested capital would eventually pay itself back if sales were affected positively by the actions taken.

Additional observations concerning HSC Chemistry Sim development that were made during the course of this work include the need to revise the unit operation development manual (Sim DLL Unit Manual presented in Appendix B). Equally, appropriate IHSC and IHSCStream documentation ought to be made available to the HSC Unit developers. Without it, the development work will partially rely on guesswork.

Work on the IHSC documentation was commenced, but not finalized. Some attention should be paid to the naming of its methods in the future. Currently, the naming schema forms a minor usability hindrance, since some of the method names do not provide proper, if any, clues as to the functionality they offer. Added readability benefits both in-house and third party development work.

In addition, the template used in HSC Unit development should be made available in C# as it would serve customers better. As discussed in Section 3.2.2, code written in Visual Basic and C# will eventually be compiled into an intermediate language, thus making the choice of implementation language a matter of mere personal preference. Because of its Java-like syntax, the chances are that the developer is more familiar with C# than with VB.

The last four items in Table 5.1 would entail a need for further research for those involved in projects that incorporate both HSC Chemistry Sim and CAPE-OPEN. At the outset of a project of that type, this thesis can easily serve as a condensed first read concerning the CAPE-OPEN world, and the accompanying list of references provides a selected list of useful resources that can be utilized in future development work.

REFERENCES

Adams, R.A. and Essex, C. (2010) *Calculus: A Complete Course* (7th Edition). Pearson Education Canada Inc. 1152 p.

AspenTech (2016) *SIMOPT CAPE-OPEN Standard*. (Online). Available at http://www.aspentech.com/engineering/SO_cape_open.aspx. (Last accessed in February 2016)

Barrett, W.M., Jr., van Baten, J. and Martin, T. (2011) Implementation of the waste reduction (WAR) algorithm utilizing flowsheet monitoring. *Computers and Chemical Engineering*, *35*, pp. 2680–2686.

Barrett, W. M., Jr. and Yang, J. (2005) Development of a chemical process modelling environment based on CAPE-OPEN interface standards and the Microsoft .NET framework. *Computers and Chemical Engineering*, *30*, pp. 191–201.

Barrett, W.M., Jr., Yang, J. and Strunjas, S. (2007) *Final Report for Verification of the Metal Finishing Facility Pollution Prevention Tool (MFFPPT)*, EPA/600/R-07/067. United States Environmental Protection Agency. 76 p.

van Baten, J.M. (2008a) Flowsheet Monitoring – How, What and Why? *Presentation at the 5th US CAPE-OPEN conference, AlChE annual meeting, Philadelphia*. (Online). Available at http://www.cocosimulator.org/downloads/AIChE_Philadelphia_2008_ExtendedSummey.pdf. (Last accessed in March 2016).

van Baten, J.M. (2008b) Flowsheet Monitoring using CAPE-OPEN. *Presentation at the 5th US CAPE-OPEN conference, AlChE annual meeting, Philadelphia*. (Online). Available at http://www.cocosimulator.org/downloads/Flowsheet_Monitoring_Philadelphia_AIChE_2008.pdf. (Last accessed in March 2016).

van Baten, J.M. (2009) Rapid prototyping of unit operation models using generic tools and CAPE-OPEN, extended abstract. *6th US CAPE-OPEN conference, AlChE annual meeting, Nashville, USA*. (Online) Available at http://www.cocosimulator.org/downloads/NashvilleExtendedAbstract.pdf. (Last accessed in April 2016).

van Baten, J. and Barrett, W.M., Jr. (2009) Technical notes on implementation of CAPE-OPEN material objects. *Training material for the 2009 European CAPE-OPEN Conference, Freising, Germany*. (Online). Available at http://www.cocosimulator.org/downloads/MaterialObjects_2009_notes.pdf. (Last accessed in January 2016).

van Baten, J. and Pons, M. (2014) CAPE-OPEN: Interoperability in Industrial Flowsheet Simulation Software. *Chemie Ingenieur Technik*, *86* (4), pp. 1052–1064.

van Baten, J. and Szczepanski, R. (2011) A thermodynamic equilibrium reactor model as a CAPE-OPEN unit operation. *Computers and Chemical Engineering*, *35*, pp. 1251–1256.

van Baten, J., Baur, R., Kooijman, H., Taylor, R., Eckert, F., Omorjan, R. and Barrett, W.M., Jr. (2015) *COCO Simulator* (Version 3.00). Computer program. Available free-of-charge at http://www.cocosimulator.org/index.html.

van Baten, J., Baur, R., Kooijman, H., Taylor, R., Eckert, F., Omorjan, R. and Barrett, W.M., Jr. (2016) *CAPE-OPEN / COCO compliancy testing*. (Online). Available at http://www.cocosimulator.org/index_compliancy.html. (Last accessed in January 2016).

Boehm, A. (2013) *Murach's Visual Basic 2012*. Mike Murach & Associates, Inc., USA. 876 p.

Cabezas, H., Bare, J.C., and Mallick, S.K. (1997) Pollution prevention with chemical process simulators: The generalized waste reduction (WAR) algorithm. *Computers and Chemical Engineering*, *21*, pp. 305–310.

CAPEC (2016a) *CAPEC Software – List of CAPEC Software*. CAPEC. (Online) Available at http://www.capec.kt.dtu.dk/Software/CAPEC-Software. (Last accessed in March 2016).

CAPEC (2016b) *Introduction to ICAS & tools*. CAPEC. (Online) Available at http://www.capec.kt.dtu.dk/documents/software/tutorials/rgani-icas-overview-2014.pdf (Last accessed in March 2016).

Chauhan, S. (2014) *Understanding .Net Framework 4.5 Architecture*. (Online) Available at: http://www.dotnet-tricks.com/Tutorial/netframework/NcaT161013-Understanding-.Net-Framework-4.5-Architecture.html. (Last accessed in April 2016).

CO-LaN (1999a) *CAPE-OPEN Next Generation Computer Aided Process Engineering Open Simulation Environment, Synthesis Report*. CAPE-OPEN. (Online) Available at http://www.colan.org/Specifications/v10/00_CO_Public_Synthesis.pdf.

CO-LaN (1999b) *CAPE-OPEN Interface specification: Numerical Solvers, version 1*. CAPE-OPEN. (Online) Available at http://www.colan.org/Specifications/v10/06_CO_Solvers.pdf.

CO-LaN (2000) *Conceptual Design Document (CDD2) for CAPE-OPEN – Interim Report – Project BE 3512, ver. 4*. CAPE-OPEN. (Online) Available at http://www.colan.org/Specifications/v10/02_CO_ Conceptual_Design_Document_CDD2.pdf.

CO-LaN (2003a) *Methods & Tools Integrated Guidelines, ver. 1*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Specifications/v10/Methods&Tools_Integrated_Guidelines.pdf.

CO-LaN (2003b) *Open Interface Specification: Collection Common Interface, version 2*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Spec%2010/Collection%20Common%20Interface.pdf.

CO-LaN (2003c) *Open Interface Specification: Identification Common Interface*, *version 3*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Spec%2010/Identification%20Common%20Interface.pdf.

CO-LaN (2003d) *Open Interface Specification: Utilities Common Interface*, *version 2*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Spec%2010/Utilities%20CommonInterface.pdf.

CO-LaN (2003e) *Open Interface Specification: Parameter Common Interface*, *version 5*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Spec%2010/Parameter%20Common%20Interface.pdf.

CO-LaN (2003f) *Open Interface Specification: Persistence Common Interface*, *version 2*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Spec%2010/Persistence%20Common%20Interface.pdf.

CO-LaN (2003g) *Open Interface Specification: Error Common Interface*, *version 7*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Spec%2010/Error%20Common%20Interface.pdf.

CO-LaN (2003h) *Open Interface Specifications: Chemical Reactions Interface*, *version 2*. CO-LaN Consortium. (Online) Available at http://www.colan.org/Specifications/v10/Chemical_Reactions_Interface_Specification.pdf.

CO-LaN (2006) *.NET Interoperability Guidelines*, *version 0.70*. CO-LaN Consortium. (Online) Available at http://www.colan.org/News/Y06/Interoperability.pdf.

CO-LaN (2011) *Unit Operations*, *version 1.0.6.25*. CO-LaN Consortium. (Online) Available at http://www. colan.org/Specifications/v10/CO_Unit_Operations_v6.25.pdf.

CO-LaN (2014) *CO (COM) Tester Suite*. Computer program. CO-LaN Consortium. (Online) Available at http:// www.colan.org/index-10.html. (Last accessed in February 2016).

Crause, C. (2013) *Object Pascal CAPE-OPEN wizard*. Computer program. (Online) Available at http://capeopenwizard.sourceforge.net/. (Last accessed in February 2016).

Edwards, J. (2013) *Process Simulation Dynamic Modelling & Control* (1st ed.). P & I Design Ltd, UK. 223 p.

Fermeglia, M., Longo, G. and Toma, L. (2008) COWAR: A CAPE OPEN software module for the evaluation of process sustainability. *Environmental Progress*, *27* (3), pp. 373–382.

Fermeglia, M. and Parenzan, M. (2007) CAPE OPEN interface revisited in terms of class-based framework: an implementation in .NET. *International Conference on Chemical & Process Engineering (ICheaP-8)*. Ischia, Italy, June 24–27, 2007. Available at http://www.mose.units.it/doc/p0359.pdf (Last accessed in January 2016).

Hietala, M. (2013) *Development of the dynamic simulation of the HSC Sim program and creation of a thermodynamic model for Peirce-Smith converter*. M.Sc. thesis. Tampere University of Technology, Master's Degree Programme in Science and Bioengineering. 72 p.

Kentala, J.-P. (2015) *Unit DLL – Unit communication and user interface*. Internal memo. Outotec Oyj, Finland.

Kentala, J.-P. (2016) *Unit DLL technical documentation*. Internal memo. Outotec Oyj, Finland.

Kentala, J.-P. and Lamberg, P. (2011) *HSC Chemistry 7.1 User's Guide. Sim Flowsheet Module – DLL Units*. Internal memo 09006-ORC-J. Outotec Oyj, Finland. 23 p.

Lajmi, A., Cauvin, S. and Ziane, M. (2009) A Software Factory for the Generation of CAPE-OPEN compliant Process Modelling Components. In de Brito Alves, R.M., Oller do Nascimento, C.A, Biscaia, E.C., Jr. (eds.). *10th International Symposium on Process Systems Engineering – PSE2009*. Elsevier B.V.

Lang, Y.-D. and Biegler, L.T. (2005) Large-Scale Nonlinear Programming with a CAPE-OPEN Compliant Interface. *Chemical Engineering Research and Design*, *83* (6), pp. 718–723.

Medeiros, D., Reichert, G. and León, G. (2014a) *CAPE-OPEN*. (Online) Available at http://dwsim.inforside.com.br/wiki/index.php?title=CAPE-OPEN. (Last accessed in January 2016).

Medeiros, D., Reichert, G. and León, G. (2014b) *Phase Identification Parameter*. (Online) Available at http://dwsim.inforside.com.br/wiki/index.php?title=Phase_Identification_Parameter. (Last accessed in January 2016).

Medeiros, D., Reichert, G. and León, G. (2015a) *Property Methods and Correlation Profiles*. (Online) Available at http://dwsim.inforside.com.br/wiki/index.php?title=Property_Methods_and_Correlation_Profiles. (Last accessed in January 2016).

Medeiros, D., Reichert, G. and León, G. (2015b) DWSIM – Process Simulation, Modeling and Optimization Technical Manual. *DWSIM Documentation*. 42 p.

Medeiros, D., Reichert, G. and León, G. (2015c) *DWSIM – Open Source Process Simulator* (Version 3.5 Build 5800). Computer program. Available free-of-charge at http://sourceforge.net/projects/dwsim/.

Microsoft (2016a) *Regasm.exe (Assembly Registration Tool)*. Microsoft Developer Network. (Online) Available at https://msdn.microsoft.com/en-us/library/tzat5yw6(v=vs.110).aspx. (Last accessed in April 2016).

Microsoft (2016b) *Visual Studio IDE*. Microsoft Developer Network. (Online) Available at https://msdn.microsoft.com/en-us/library/dn762121.aspx. (Last accessed in February 2016).

Microsoft (2016c) *What is a DLL?* Microsoft Support. (Online) Available at https://support.microsoft.com/en-us/kb/815065. (Last accessed in February 2016).

Microsoft (2016d) *Runtime Callable Wrapper*. Microsoft Developer Network. (Online) Available at https://msdn.microsoft.com/en-us/library/8bwh56xe(v=vs.110).aspx. (Last accessed in January 2016).

Microsoft (2016e) *COM Callable Wrapper*. Microsoft Developer Network. (Online) Available at https://msdn.microsoft.com/en-us/library/f07c8z1c(v=vs.110).aspx. (Last accessed in January 2016).

Microsoft (2016f) *COM Wrappers*. Microsoft Developer Network. (Online) Available at https://msdn.microsoft.com/en-us/library/5dxz80y2(v=vs.110).aspx. (Last accessed in January 2016).

Moler, C.B. (2004) *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics. Philadelphia, USA. 336 p.

Morales-Rodríguez, R., Gani, R., Déchelotte, S., Vacher, A. and Baudoin, O. (2008) Use of CAPE-OPEN standards in the interoperability between modelling tools (MoT) and process simulators (Simulis Thermodynamics and ProSimPlus). *Chemical Engineering Research and Design*, *86*, pp. 823–833.

Morton, W. (2003) Equation-oriented simulation and optimization. *Proceeding of the Indian National Science Academy*, *69*, *A*, *3&4*, pp. 317–357.

Nielsen, J. (1995) *10 Usability Heuristics for User Interface Design*. (Online) Available at https://www.nngroup.com/articles/ten-usability-heuristics/. (Last accessed in March 2016).

Outotec (2015) *Outotec HSC Chemistry 8 Brochure*. Outotec Oyj, Finland. 12 p. (Online) Available at http://www.outotec.com/ImageVaultFiles/id_1612/cf_2/OTE_Outotec_HSC_Chemistry_8_eng.PDF. (Last accessed in February 2016).

Outotec (2016a) *HSC Version Summary (History)*. Outotec Oyj. (Online) Available at http://www.outotec.com/en/Products--services/HSC-Chemistry/History/. (Last accessed in February 2016).

Outotec (2016b) *Outotec > About us > History*. Outotec Oyj. (Online) Available at http://www.outotec.com/en/About-us/History/. (Last accessed in February 2016).

Outotec (2016c) *Sim – Process Simulation*. Outotec Oyj. (Online) Available at http://www.outotec.com/en/Products--services/HSC-Chemistry/Calculation-modules/Sim--process-simulation/. (Last accessed in February 2016).

Peng, D.-Y. and Robinson D.B. (1976) A New Two-Constant Equation of State. Industrial and Engineering Chemistry Fundamentals, *15* (1), pp. 59–64.

Pons, M. (2003) The CAPE-OPEN Interface Specification for Reactions Package. In Kraslawski, A. and Turunen, I. (eds.). *European Symposium on Computer Aided Process Engineering – 13*. Elsevier Science B.V.

Pons, M.E. (2005) CAPE-OPEN Consultancy Scheme. In Puigjaner, L. and Espuña, A. (eds.). *European Symposium on Computer Aided Process Engineering – 15*. Elsevier B.V.

ProSim (2016) *ProSimPlus. Steady-state simulation and optimization of processes.* (Online) Available at http://www.prosim.net/en/software-prosimplus--1.php. (Last accessed in April 2016)

Remes, A. and Kentala, J.-P. (2014) *HSC Sim 8 unit model programming manual – DLL interface specification.* Internal memo 14012-ORC-M. Outotec Oyj, Finland. 10 p.

Schopfer, G., Wyes, J., Marquardt, W. and von Wedel, L. (2005) A Library for Equation System Processing based on the CAPE-OPEN ESO Interface. In Puigjaner, L. and Espuña, A. (eds.). *European Symposium on Computer Aided Process Engineering – 15.* Elsevier B.V.

Sheldon, B., Hollis, B., Windsor, R., McCarter, D., Hillar, G.C. and Herman, T. (2013) *Professional Visual Basic 2012 and .NET Programming.* John Wiley & Sons, Inc., Indianapolis, USA. 912 p.

Soares, R.P. and Secchi, A.R. (2004) Modification, simplification, and efficiency tests for the CAPE-OPEN numerical open interfaces. *Computers and Chemical Engineering*, *28*, pp. 1611–1621.

Stallings, R. (2005) *Operating Systems: Internals and Design Principles* (5th ed.). Pearson Education, Inc., USA. 818 p.

Testard, L. and Belaud, J.P. (2005) A CAPE-OPEN framework for process simulation solutions integration. In Puigjaner, L. and Espuña, A. (eds.). *European Symposium on Computer Aided Process Engineering – 15.* Elsevier B.V.

Thomas, I. (2011) A Process Unit Modelling Framework within a Heterogeneous Simulation Environment. In Pistokopoulos, E.N., Georgiadis, M.C. and Kokossis, A. (eds.). *21st European Symposium on Computer Aided Process Engineering – ESCAPE21.* Elsevier B.V.

Urroz, G.E. (2004) Solution of non-linear equations. (Online) Available at http://ocw.usu.edu/Civil_and_Environmental_Engineering/Numerical_Methods_in_Civil_Engineering/NonLinearEquationsMatlab.pdf. (Last accessed in April 2016).

Young, D.M. and Cabezas, H. (1999) Designing sustainable processes with simulation: the waste reduction (WAR) algorithm. *Computers and Chemical Engineering*, *23*, pp. 1477–1491.

Young, D.M., Scharp, R. and Cabezas, H. (2000) The waste reduction (WAR) algorithm: Environmental impacts, energy consumption, and engineering economics. *Waste Management*, *20*, pp. 605–615.

APPENDICES

Appendix A. Excerpt from the IHSC interface specifications document. 1 page.

Appendix B. Sim DLL Units Manual. 25 pages.

Appendix C. HSC Chemistry Sim unit operation template (Visual Basic). 1 page.

Appendix D. Code for a sample unit operation, the Perfect Mixer (Visual Basic). 1 page.

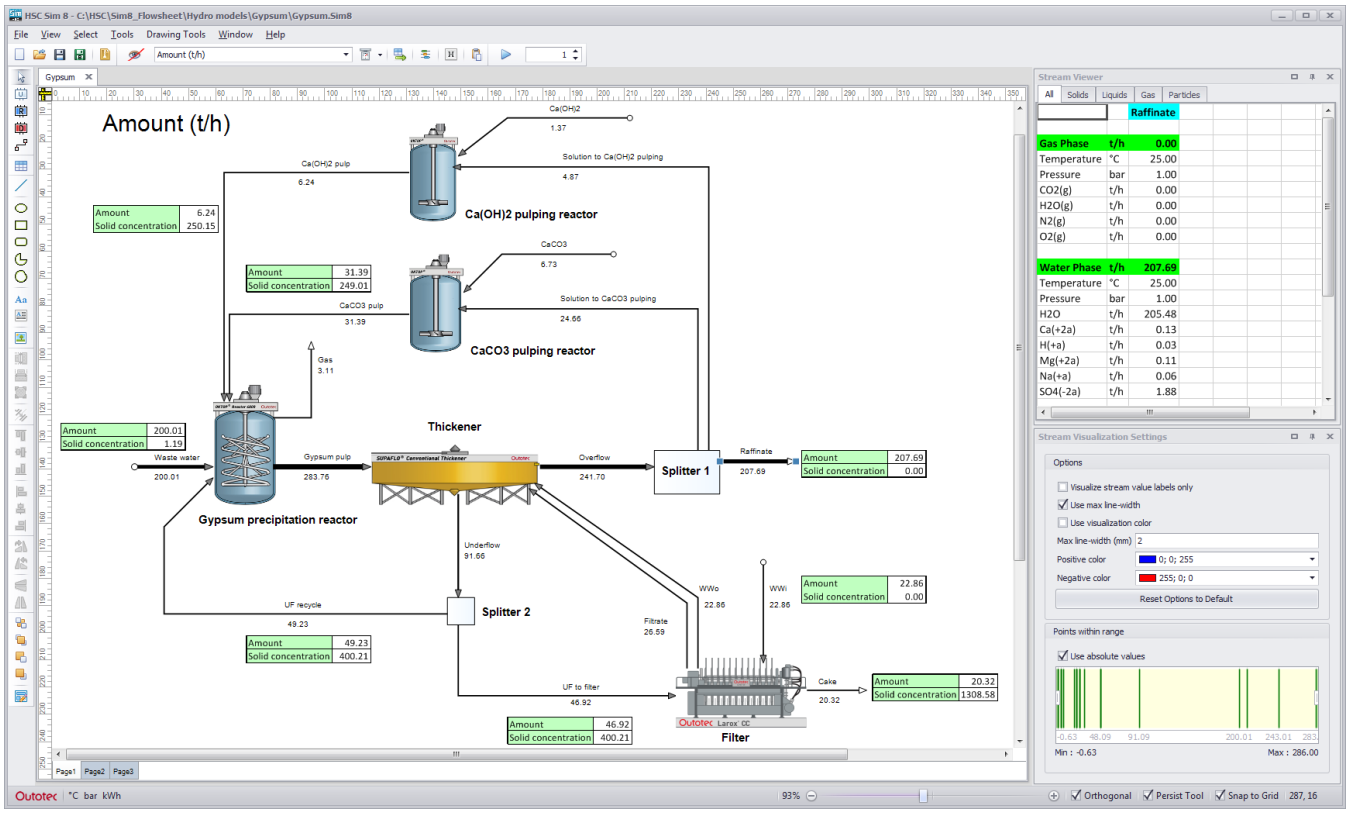
Appendix E. Ten general principles of the interaction design. 1 page.

Species (G, H, S, C_p)

Function	Return type / Description
HCM(ByVal Species As String, ByVal T As Double)	Double Enthalpy (per Nm³) of species at given temperature (may throw an HSCException)
HLat(ByVal Species As String, ByVal TemperatureInUserUnits As Double)	Double Enthalpy excluding phase transformations (per mol) of species at given temperature
HLatTP(ByVal Species As String, ByVal TemperatureInUserUnits As Double, ByVal PressureInUserUnits As Double)	Double Enthalpy excluding phase transformations (per mol) of species at given temperature. Tries to take pressure into account (doesn't work for every species, in such case normal Hlat is used)
HNm3(ByVal Species$, ByVal TemperatureInUserUnits#)	Double Enthalpy (per Nm³) of species at given temperature (HCM function is recommended)
HSCatT(ByVal species As String, ByRef temperatureInKelvins As Double, ByRef deltaHf_kcal As Double, ByRef deltaH_kcal As Double, ByRef S_kcal As Double, ByRef Cp_kcal As Double, ByRef deltaG_kcal As Double, ByRef molWeight As Double, ByRef density As Double, ByRef reference As String, Optional NoCrissCobble As Boolean = False)	Boolean Parameters: Species = Input: Species name ("H2O", "Mg", etc.), temperatureInKelvins = Input: Temperature in K, deltaHf_kcal = Output: Formation enthalpy - calculates only the enthalpy of formation at specific temperature (kcal), deltaH_kcal = Output: Total enthalpy change - calculates the enthalpy of formation and the enthalpy of heating to specific temperature (kcal), S_kcal = Output: Absolute entropy (kcal), Cp_kcal = Output: Heat capacity (kcal), deltaG_kcal = Output: Change of Gibbs energy (kcal), MolWeight = Output: Molar weight (g)
HSCatTP(ByVal Species As String, ByRef tempInKelvins As Double, ByVal P As Double, ByVal criticalTemperatureInKelvins As Double, ByVal criticalPressure As Double, ByVal phase As String, ByRef deltaHf_kcal As Double, ByRef deltaH_kcal As Double, ByRef S_kcal As Double, ByRef Cp_kcal As Double, ByRef deltaG_kcal As Double, ByRef molWeight As Double, ByRef density As Double, ByRef REF$, Optional noCrissCobble As Boolean = False)	Sub. Parameter: deltaHf_kcal = Output: Formation enthalpy - calculates only the enthalpy of formation at specific temperature, deltaH_kcal = Output: Total enthalpy change - calculates the enthalpy of formation and the enthalpy of heating to specific temperature, S_kcal = Output: Absolute entropy (kcal), Cp_kcal = Output: Heat capacity (kcal), deltaG_kcal = Output: Change of Gibbs energy (kcal), MolWeight = Output: Molar weight (g)
Solu(ByVal Species As String)	Double The solubility of a species in H2O

M. Leino, A. Remes, J-P. Kentala

52. Sim DLL Units Manual



HSC Sim flowsheet process model consist of unit operations connected with each other using streams. Two types of unit operation models may be used: 1) Excel and 2) DLL type models.

Excel models may be created using build in Excel editor and unit wizards.

DLL type unit models must be created using some of MS Visual Studio languages. This manual describes how to create your own models using VB.NET and build them into a DLL file used in Sim process models.

Contents

52.1. Introduction

The HSC Chemistry 8 flowsheet simulator module Sim offers its users the possibility of creating their own custom unit operation models. Unit operation models can be created by using either a built-in Excel editor or by using one of the programming languages supported by Microsoft Visual Studio. These self-made unit operation models enable the user to create tailored process models for many different types of industrial processes.

This document helps write the code for a custom unit by using Microsoft Visual Studio IDE (Integrated Development Environment) and Visual Basic.NET programming language. Although we use Visual Studio and Visual Basic.NET in the examples, this is not the only suitable programming language. Other options include for instance C# and F# (see http://en.wikipedia.org/wiki/List_of_CLI_languages for further information). Readers with no previous knowledge of these languages might find it useful later to refer to some online or printed learning resources, which are readily available.

To make the instructions as easy to understand as possible, a simple functioning model for a thickener unit will be created. Note the unit files are sometimes referred to as **DLL units** as the models are wrapped inside dynamic link library (DLL) files.

52.2. Requirements

As mentioned earlier, the tool of choice for development in this manual is Visual Studio. A free version of Visual Studio (Visual Studio Express for Windows Desktop) is available for download from Microsoft for free, but it is possible to use any other version of it (Professional, Premium or Ultimate). Version 4.5 (or later) of the .NET framework has to be installed in the system when developing DLL unit models. For example, Visual Studio 2012 or newer includes installation files for .NET Framework 4.5.

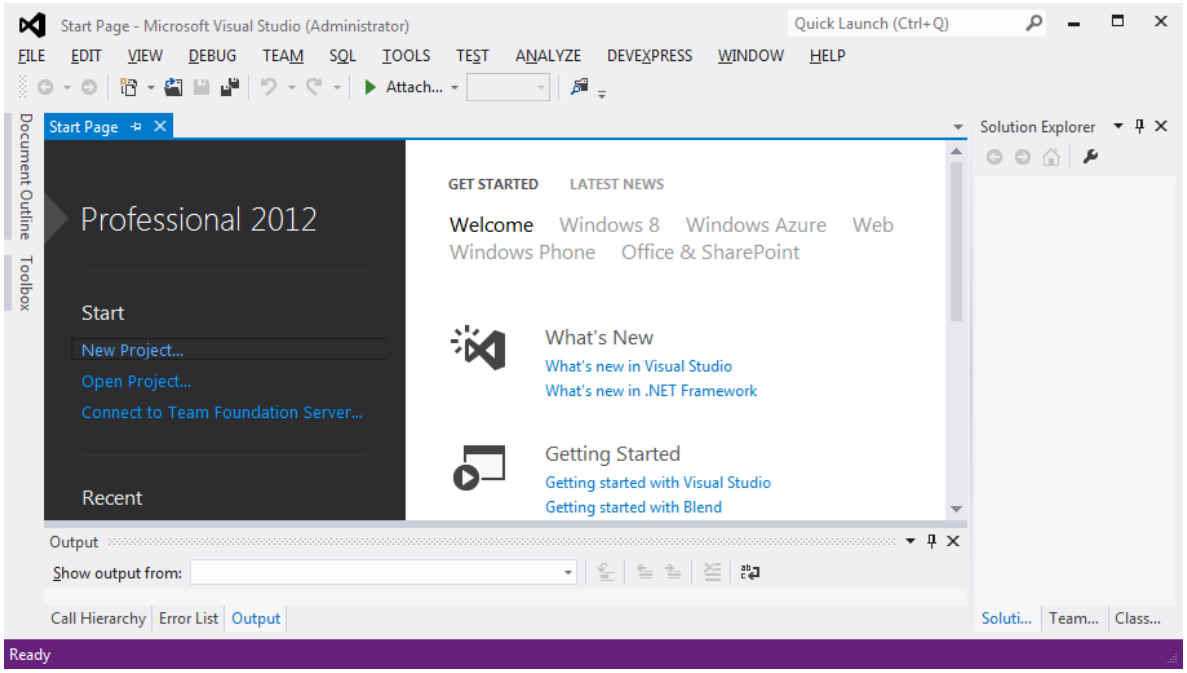


Figure 1. The examples in this document were written with Visual Studio 2012 Professional. The other editions of Visual Studio look similar, but the menu arrangements vary depending on the user settings.

52.3. Creating a new project

Open the New Project dialog box by choosing File > New > Project. Open Installed | Templates | Visual Basic | Windows group as shown in Figure 2. Choose the Class Library template. Next, enter the location (folder) and name for the project. The name should have the following structure: *YourUnitName.HSC8Addin*, as this later makes HSC Sim register the unit appropriately for use. The example file is named ExampleUnit.HSC8Addin.

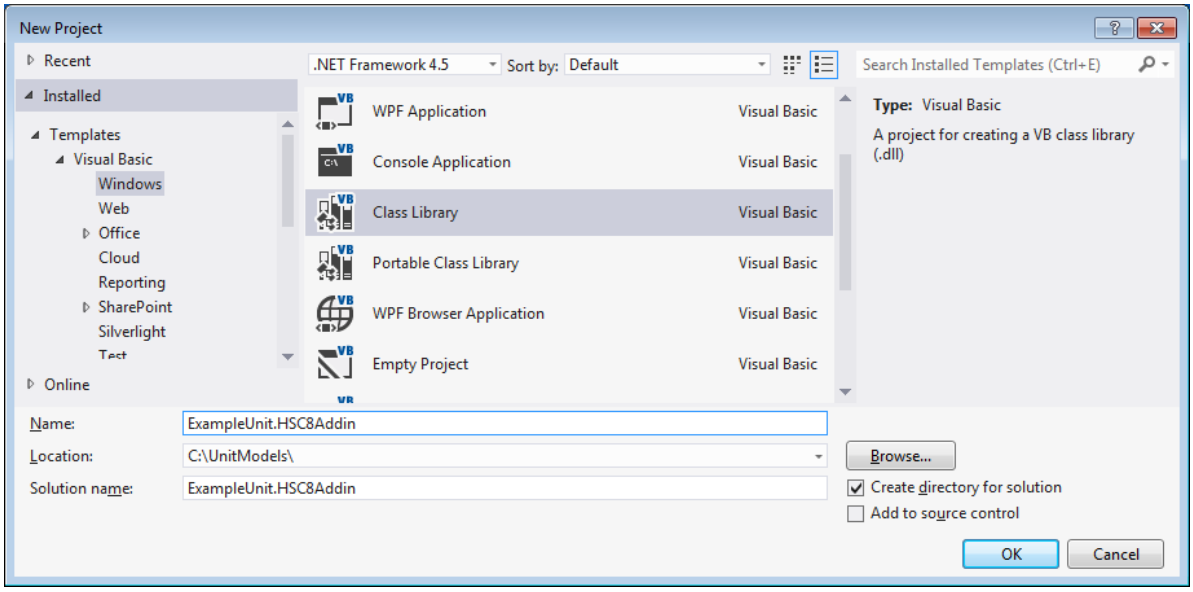


Figure 2. New Project dialog box.

Next, choose Project > YourUnitName.HSC8Addin Properties… to open the project properties dialog. Open the References tab, see Figure 3.

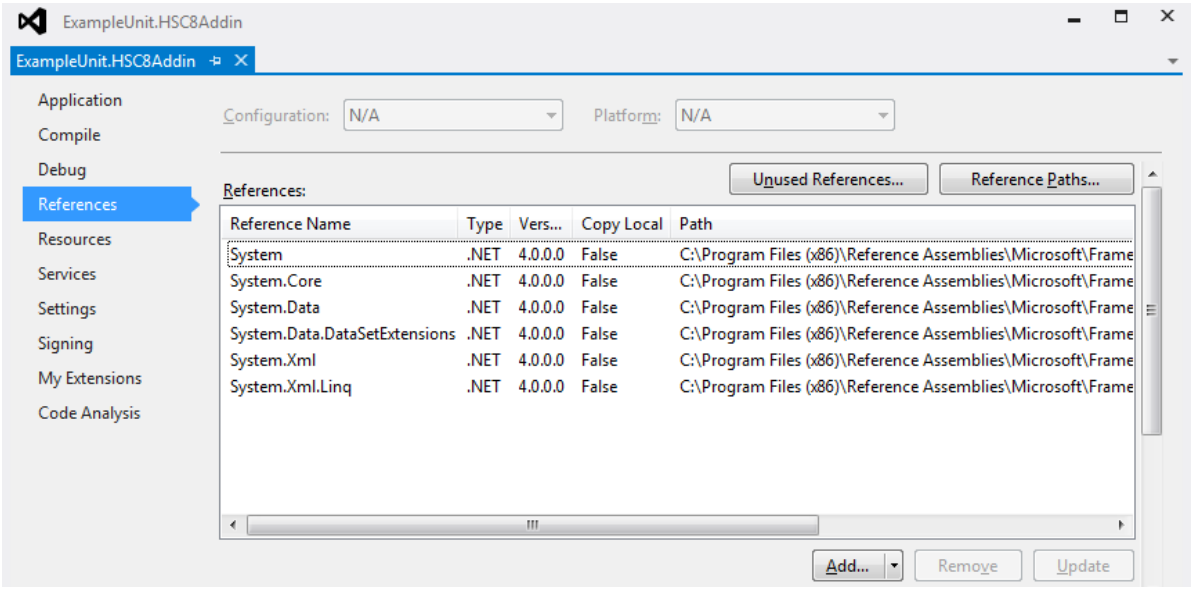


Figure 3. References tab in the Project Properties panel.

Press the Add button and select the following assembly from the Assemblies | Framework group by ticking its check box and clicking OK.

```
System.ComponentModel.Composition (version 4.0.0.0)
```

Continue adding references by clicking Browse at the bottom of the window and locating the following HSC Chemistry assemblies in the HSC8/Programs folder:

```
HSCExternalUnitDLL.dll
Outotec.HSC.Common.dll
Outotec.HSC.HSCStream.dll
```

Add the assemblies to the project by clicking Add. After successful addition of the libraries, your References tab should look similar to Figure 4.

(Note: the above may have changed after this manual was completed. The idea was to modify the framework around the unit models so that only one reference would be needed instead of many.)

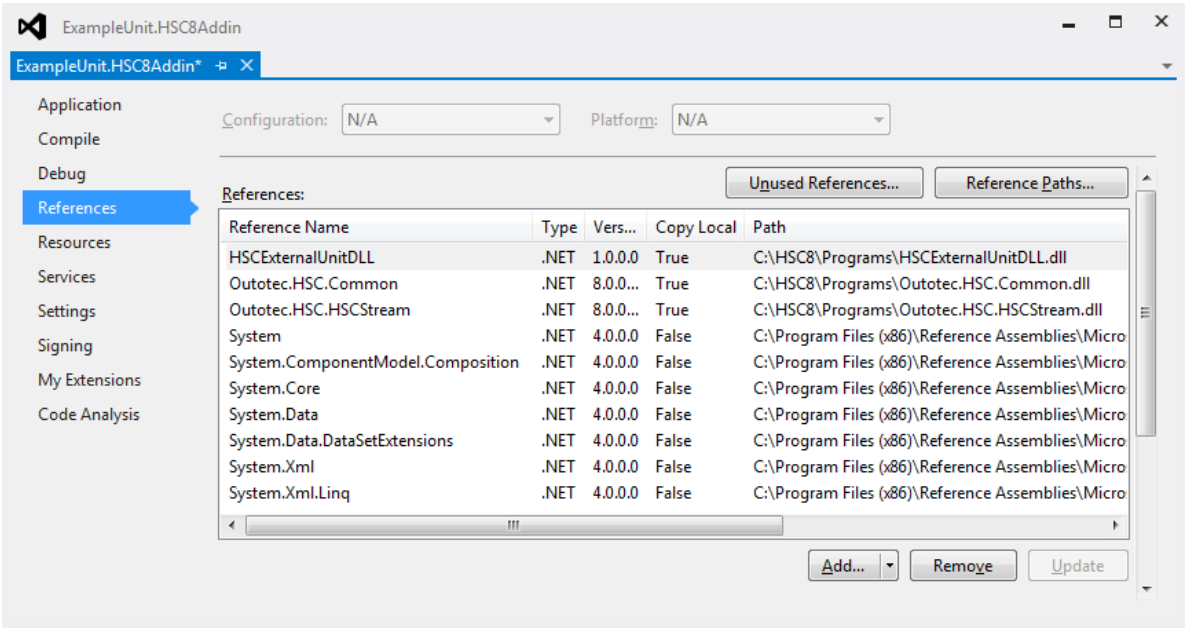


Figure 4. References tab after adding the references to the required class libraries.

The references need to be added to the project, as they contain classes that will be utilized in programming the unit models. In the example project, we do not use any of the mineral processing classes. When creating a mineral processing unit model, add a reference to the following library (see Appendix A)

```
Outotec.HSC.HSCStream.Minerals.dll
```

The project properties tab can be closed now.

52.4. DLL Unit Template

DLL unit VB.NET template may be used to create new DLL units. The user must specify Namespace, Input and output streams, model parameters, internal states, runtime variables and static or dynamic calculation formulas.

When model is ready user must compile DLL and copy DLL file into \HSC8\Programs folder. Later on DLL unit files will have own folder. Then user can start to use DLL model in Sim flowsheet.

The parameters which user specifies in the parameters section will arise automatically into Parameters sheet in the same order as typed in the source code.

The following chapters will describe these steps in more detail.

```
Imports Outotec.HSC.HSCStream
Imports System.Collections.Generic
Imports Outotec.HSC.Common
Imports Outotec.HSC.HSCStream.Minerals
Imports System.Math
Imports Outotec.HSC.HSCUnitDLL.MinProUnitFunctions
Imports Outotec.HSC.HSCExternalUnitDLL

<HSCUnitDLL("Name",
    "Description.",
    "© Outotec (Finland), Authors: Antti Remes",
    HSCUnitDLLAttribute.Mode.StaticOnly,
    TypeCode:="MU-100-10",
    Technology:="Others")>
Public Class _Template
    Inherits UnitDLL

#Region "Stream Configuration"

    '---Inputs:

    <StreamIn("Input", 1)>
    Public Property Input As IHSC8Stream

    '---Outputs:

    <StreamOut("Output", 1)>
    Public Property Output As IHSC8Stream

#End Region

#Region "Model Parameters"
    'Not existing for this unit.
#End Region

#Region "Model Internal States"
    'Not existing for this unit.
#End Region

#Region "Runtime Variables"
    'Not existing for this unit.
#End Region
```

```
#Region "Static and Dynamic Calculation"

    Protected Overrides Sub CalcStatic(ByVal HSC As IHSC8)

    End Sub

    Protected Overrides Sub CalcDynamic(ByVal HSC As IHSC8, ByVal dynamicInfo As
IDynamicCalculationInfo)

    End Sub

#End Region

#Region "Internal Procedures"

#End Region
End Class
```

Figure 5. Image of DLL Unit Template.

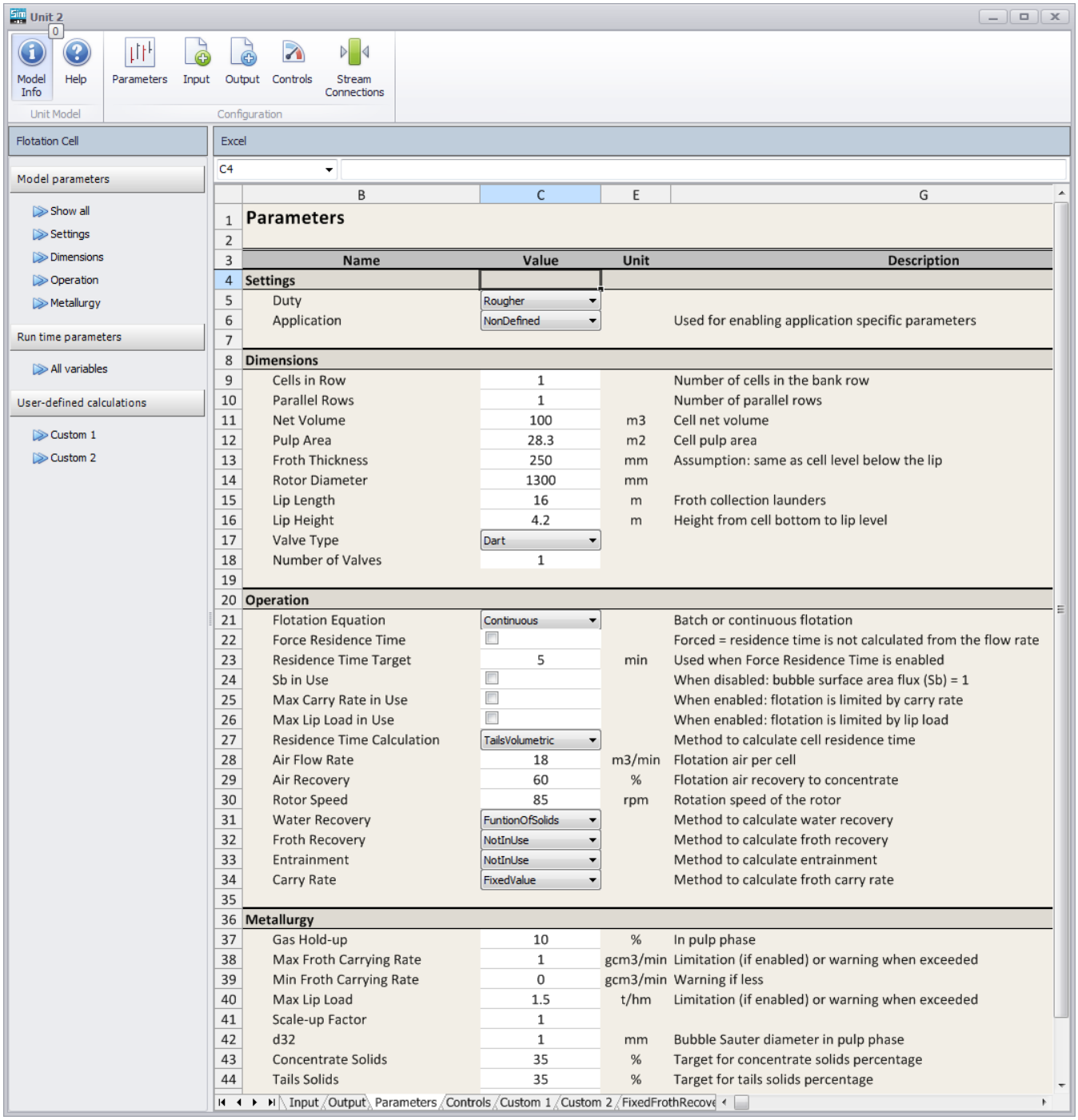


Figure 6. DLL Unit source code #Region "Model Parameters" will fill automatically Parameters sheet in DLL unit editor workbook. Printscreen example is from Flotation cell unit model with long list of different type of calculation parameters.

52.5. Namespaces

The references have made the existing HSC namespaces available in our class library project. To make it easier to refer to the classes in those namespaces, we will add a few `Imports` statements to the beginning of the file. This makes referring to the classes easier. As an example, a variable declaration such as

```
Dim sb as New System.Text.StringBuilder()
```

can be written as

```
Dim sb as New StringBuilder()
```

after the appropriate namespace (in this case System.Text) has been imported.

As a result, your code should look somewhat similar to the code in Figure 7.

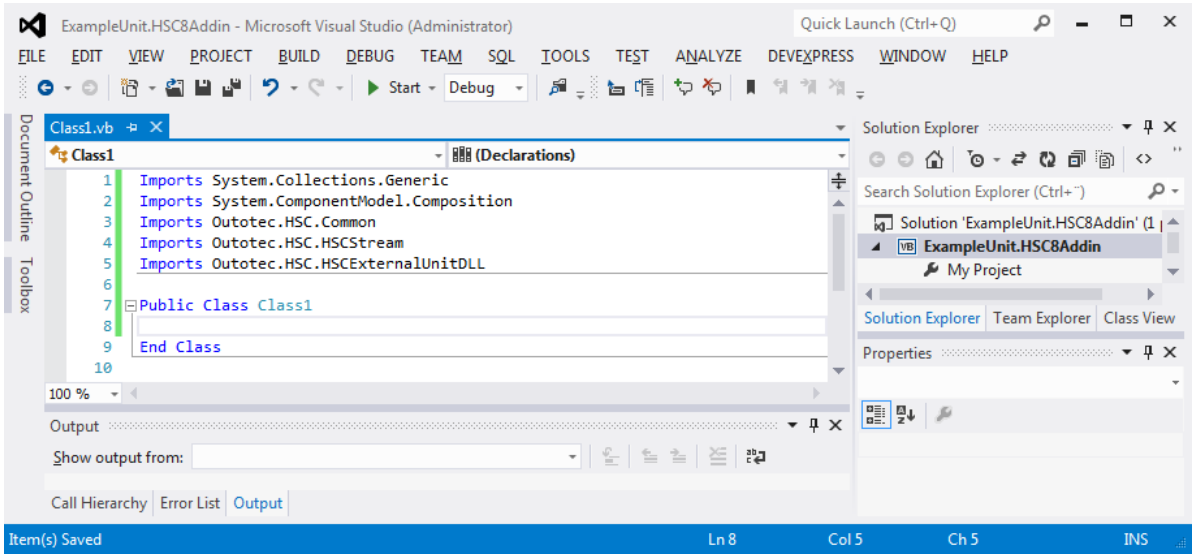
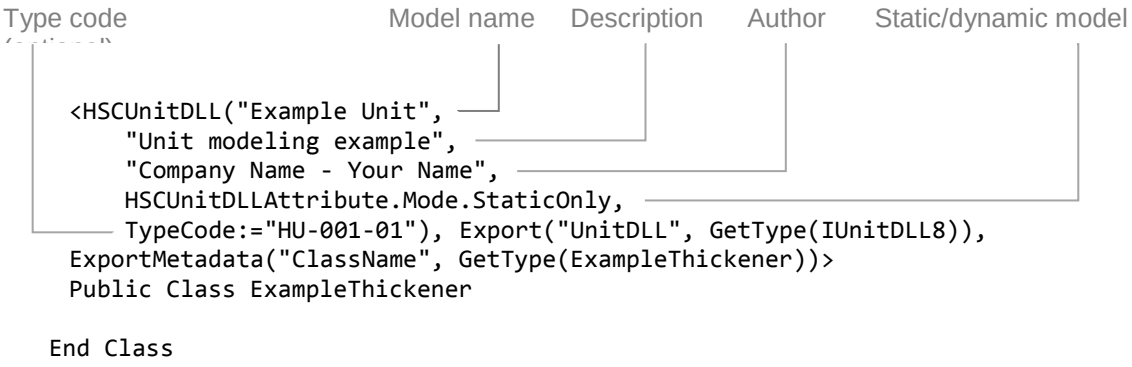


Figure 7. `Imports` statements have to be included at the beginning of the source code file.

From here on, screenshots are *not* used to show how the source code has been modified; instead they are shown as text. Copying and pasting a code to your own model's source code is OK, but you might find it useful to know that the complete code for the example unit is listed in Appendix B.

Instead of using a generic class name such as Class1 for our class, we will change the class name to ExampleThickener. The class file name can be changed in the solution explorer by right-clicking the file name and choosing Rename (e.g. ExampleThickener.vb). Keeping the names consistent might turn out to be useful later, especially if you have several files and classes in your class library project. It is possible to create several unit model classes within one file, but a more common practice is to have each class in its own file.

The next step is to add general information about your unit. This is done by using an attribute element (marked as <*attribute info*>) that resides just before the `class` statement, see the example below. The information inserted here is shown in the 'select unit models' dialog in HSC Sim.
.




```
<HSCUnitDLL("Example Unit",
    "Unit modeling example",
    "Company Name - Your Name",
    HSCUnitDLLAttribute.Mode.StaticOnly,
    TypeCode:="HU-001-01"), Export("UnitDLL", GetType(IUnitDLL8)),
ExportMetadata("ClassName", GetType(ExampleThickener))>
Public Class ExampleThickener

End Class
```

The information contained in the attribute includes the name of the model, description of the model, the name of the author, and whether the model is intended for static or dynamic modeling (at the moment only Static can be used). An optional field in the attribute is the type code field. The type code can be any user-selected code to identify the model in addition to its name. For example, the HSC Sim models use the following notation: first, a two-lettered abbreviation that tells us the category of the unit (MU – mineral unit, HU – hydro unit, PU – pyro unit, RU – recycling unit), followed by a three-digit process area number and a two-digit unit model number. The information within the required attribute fields is important for HSC Sim to be able to handle the file properly, so it should be written in the same manner as in the example.

The base class for each model class is the UnitDLL class. It has to be inherited and the Visual Basic way to express this is to use the `Inherits` keyword right after your class name.

```
Public Class ExampleThickener
    Inherits UnitDLL

#Region "Static and Dynamic Calculation"

    Protected Overrides Sub CalcStatic(ByVal HSC As IHSC8)
        CalcOutput()
    End Sub

    Protected Overrides Sub CalcDynamic(ByVal HSC As IHSC8,
        ByVal DynamicInfo As IDynamicCalculationInfo)
        CalcOutput()
    End Sub

#End Region

End Class
```

When the text Inherits UnitDLL is inserted, Visual Studio automatically adds two methods (CalcStatic and CalcDynamic), that have to be overridden in the model class. In this example, an internal Private Sub method called CalcOutput has been added. It is called by both the static and dynamic calculation modes. If the model is static only, both functions, CalcStatic and CalcDynamic, have to perform the same static calculation just like in the example.

The code can include #Region directives, which make it is possible to collapse and hide blocks of code in Visual Basic files. It is not compulsory to use the #Region directive, but it might be useful to be able to hide blocks of code in larger files. This way it is easier to focus on only portions of it at a time.

52.6. Input and output streams

In HSC Sim 8, material and energy are transferred from one unit to another via streams. The examples below show how to declare the streams that connect to a unit in HSC Sim.

A unit can have input and output streams in the following ways:

- One or more input streams (the number of streams is known). Each of them is declared as a Property preceded by a StreamIn attribute, as shown here:

```
<StreamIn("Feed", 1)>
    Public Property Feed As IHSC8Stream
```

- One or more input streams (the number of streams is unknown)

```
<StreamIn("Feeds", 1)>
    Public Property Feeds As List(Of IHSC8Stream)
```

- One or more output streams (the number of streams is known)

```
<StreamOut("Overflow", 1)>
    Public Property Overflow As IHSC8Stream
```

- Several outputs, the number of outputs is not pre-defined

```
<StreamOut("Outputs", 1)>
Public Property AllOutputs As List(Of IHSC8Stream)
```

- Optional inputs (when it is not actually obligatory to include one)

```
<StreamIn("Cake_Wash_Water", 2, Required:=False)>
      Public Property CakeWashWater As IHSC8Stream
```

- Optional outputs (when it is not obligatory to include one)

```
<StreamOut("Water_Outlet_A", 3, Required:=False)>
      Public Property WaterOutA As IHSC8Stream
```

Our example unit has one input stream and two output streams. The input stream is called *Feed* and is assigned a stream number of 1 and set as the default input. If an input stream is defined as default, HSC Sim automatically assigns the first user-drawn input stream to this input stream. If no default streams are set, HSC Sim opens a dialog box in which the latest user-drawn stream can be assigned to one of the pre-defined streams in the model (such as Overflow or Underflow, as below).

```
#Region "Stream Configuration"

    'Inputs:

    <StreamIn("Feed", 1, IsDefaultInput:=True)>
    Public Property Feed As IHSC8Stream

    'Outputs:

    <StreamOut("Overflow", 1)>
    Public Property Overflow As IHSC8Stream

    <StreamOut("Underflow", 2)>
    Public Property Underflow As IHSC8Stream

#End Region
```

52.7. Parameters

Model parameters have initial values and the information concerning them are defined inside the *Parameter* attribute. The heading, name, measure unit, and description have to be specified for a parameter, but its limits are optional. Users can later change the parameter values in HSC Sim 8, where a user interface is automatically provided for doing this. An example of model parameter declaration is shown here:


```
<Parameter("General", "Net Volume", "m3", "Cell net volume", GreaterThan:=0)>
Public Property NetVolume As Double = 100.0
```

In the example below, the heading is declared as a constant character string. Using constants is useful as the heading field is used to categorize and group the parameters of the model. Using constant strings also makes group naming less error-prone. Maximum and minimum limits are also set for the parameter (minimum as MinLim:=1, maximum as MaxLim:=100). A parameter value has to be set by the user between its maximum and minimum values if such values are defined.

```
#Region "Model Parameters"

    Const NODE0 As String = "PARAMETERS"

    <Parameter(NODE0, "Underflow solids", "%", "Underflow solids percentage",
        MinLim:=1, MaxLim:=100)>
    Public Property UnderFlowSolidsPercentage As Double = 50.0

#End Region
```

A brief explanation regarding the limits:

MinLim	stands for	>=
MaxLim	stands for	<=
LessThan	stands for	<
GreaterThan	stands for	>

As for the colon, e.g. MinLim:=1, this means that the value for MinLim is passed by name. In other words, it is okay to code these limits in any sequence and it is not compulsory to indicate if a particular limit is omitted (i.e. they are *optional*).

Parameter values can also be provided by using a specific table type class, see *UnitTable.vb*.

52.8. Model internal states

Model internal states are public properties, which are stored by HSC Sim in the model .xls file, but are not shown to the user in the user interface. These properties can be used to store the model state between calculations, e.g. previous round output streams can be stored.

```
<InternalState("Tails streams of the bank")>
Public Property BankTails As New List(Of IHSC8Stream)
```

Since the example does not need to store any of its states, this part can be neglected or a region stub can be inserted into the example model's source code for future reference.

```
#Region "Model Internal States"
    'Not in use in this unit
#End Region
```

52.9. Runtime variables

Runtime variables are for presenting the model internal states during the calculation, i.e. information that cannot be provided from/to the model with the output/input stream. They are also used for controlling certain model operating conditions (=model internal states) during the runtime, especially in dynamic simulations. An example of such a case is a tank level measurement (runtime output variable) that is controlled by using a HSC control that changes the tank outlet valve position (runtime input variable).

The variables can be inputs (write only), outputs (read only), or inputs and outputs (read/write).

```
<RuntimeVariable("P", "kPa", "Pressure drop", False, True)>
Public Property Pressure As Double
```

The example does not utilize runtime variables either, so only a region stub is inserted into to the code, although it is not necessary. In the future it will be easy to make amendments to the code by filling the empty regions with relevant program code.

```
#Region "Runtime Variables"
    'Not in use in this unit
#End Region
```

It is not mandatory to specify the "write-only" or "read-only" setting in the RuntimeVariable attribute. HSC Sim can detect the setting also from the property ReadOnly and/or WriteOnly modifiers:

```
 <RuntimeVariable("Level", "m", "Some level")>
Public ReadOnly Property Level As Double ' Automatically ReadOnly (unit DLL
output)

<RuntimeVariable("Valve setting", "m", "Valve setting")>
Public WriteOnly Property Pressure As Double ' Automatically write only (unit
DLL input)
```

```
<RuntimeVariable("Angle", "Degrees", "")> ' Automatically read/write (both input
and output)
Public Property Angle As Double
```

52.10. Model parameter initialization

`InitModelParameters` sub is called by HSC Sim when the model is applied. This sub is not necessary for all units. Typically this is needed if some tables are to be initialized based on the input stream structure of the model.

```
Protected Overrides Sub InitModelParameters(HSC As IHSC8, RefFeed As
IHSC8Stream)
          'MyBase.InitModelParameters(HSC, RefFeed)
End Sub
```

The example code does not need to initialize any parameters beforehand. The example code is left without any reference to model parameter initialization.

52.11. Calculation within the example unit model

The simple thickener unit that is modeled here first counts the amount of solids and water in the input stream. It then calculates how much water is needed in the underflow to reach the desired solids percentage (set via a parameter) in the underflow stream. Solids percentage *x* in the underflow is calculated as follows:

$$\frac{m_{solids}}{m_{solids} + m_{water}} = x \quad (1)$$

Since *x* is the parameter value that the user sets and the amount of solids is known due to prior calculations, equation (1) can be rearranged to

$$m_{water} = \frac{(1-x)}{x} m_{solids} \quad (2)$$

Equation (2) is included in the model code. Finally, the example unit directs all the solids and the required amount of water to the underflow and the excess water to the overflow.

It is worth mentioning that it is possible to send messages from the model to the HSC Sim user interface. The sent messages are shown in the log viewer. Some examples are included in the example code and the way the messages are displayed is demonstrated in the next chapter.

The Unit model can send three types of messages to the user during the calculation:

- Normal log message

```
WriteToLog("Starting calculations")
```

- Info messages

```
InfoMessage("Cyclone is roping according to SPOC criterion.")
```

- Warning messages

```
WarningMessage("Solids percentage target(s) could not be reached.")
```

- Error messages

```
RaiseFatalErrorException("Recovery calculation for selected particle
floatability type is not supported by this model version.")
```

About debugging a class library project (such as a unit model project), please refer to a web page like https://msdn.microsoft.com/en-us/library/ms164704.aspx (Debugging DLL Projects) and follow the given instructions (when you need to select *Start external program* please choose C:\HSC8\Programs\HSC-Sim.exe).

It is possible to utilize the info and warning message, too. Add info messages to your model and have it send appropriate information to the log viewer so that you can track your model operation. If unexpected or inconsistent results come up, make changes to the code and re-build your project. Then test the new version of your model within HSC Sim.

Below is the code for the CalcOutput method. It is called by the CalcStatic and CalcDynamic methods and it is the method in our example that does the calculations. It includes both an InfoMessage and a WarningMessage to show how they can be implemented. No further code is needed in this example.

```
#Region "Internal Procedures"

    Private Sub CalcOutput()

        Dim massOfSolids As Double
        Dim massOfWater As Double
        Dim requiredWater As Double
        Dim waterToOverflow As Double
        Dim x As Double = UnderFlowSolidsPercentage

        'calculate the amount of material in solid and water phases
        For Each c_amount In Feed.AllStreamComponentAmounts

            If c_amount.PhaseGroup.Phase = "Solid" Then
                massOfSolids += c_amount.Amount
            End If

            If c_amount.PhaseGroup.Phase = "Liquid" Then
                massOfWater += c_amount.Amount
            End If

        Next

        'calculate the required amount of water
        requiredWater = (100 - x) / x * massOfSolids
        waterToOverflow = massOfWater - requiredWater

        'an example of an InfoMessage
        InfoMessage("Required amount of water is " & CStr(requiredWater))

        If waterToOverflow < 0 Then
            WarningMessage("Not enough water to proceed.")
        Else

            'transfer phases to the right streams
            For Each co In Feed.AllStreamComponentAmounts

                If co.PhaseGroup.Phase = "Solid" Then
                    Underflow.AddStreamComponentAmount(co.Component,
                                                       co.PhaseGroup, co.Amount)
                End If

                If co.PhaseGroup.Phase = "Liquid" Then
                    Underflow.AddStreamComponentAmount(co.Component,
                                                       co.PhaseGroup, requiredWater)
                    Overflow.AddStreamComponentAmount(co.Component,
                                                       co.PhaseGroup, waterToOverflow)
                End If

            Next

        End If

    End Sub

#End Region
```

52.12. Applying and testing the model

Once the model is ready, choose Build > Build Solution. This command builds the solution and creates the ExampleThickener.HSC8Addin.dll file in the bin/Debug folder, which in the case of our example is located at

```
C:\UnitModels\ExampleUnit.HSC8Addin\ExampleUnit.HSC8Addin\bin\Debug
```

Copy the `YourUnitName.HSC8Addin.dll` file to the HSC8/Programs folder.
For easy and fast testing, you can set the compile target directly to C:\HSC8\Programs folder. This way you can just build the project and test it immediately in HSC Sim.

Start HSC Sim 8. Draw a generic unit and right-click on the new unit. Choose “Unit Model Editor”. Make sure that Unit 1 is selected and then locate Example Unit (since the type code for this unit started with an H, it is listed in the Reactions tab, which shows the Hydro units). Double-click the name of the unit and click OK. Now our model is applied to the newly drawn unit.

Draw an input stream in the unit. Add an output stream. A dialog box opens that asks you to assign an output port to the stream; choose Overflow. Add another stream. Now the flowsheet should look something like the one in Figure 8.

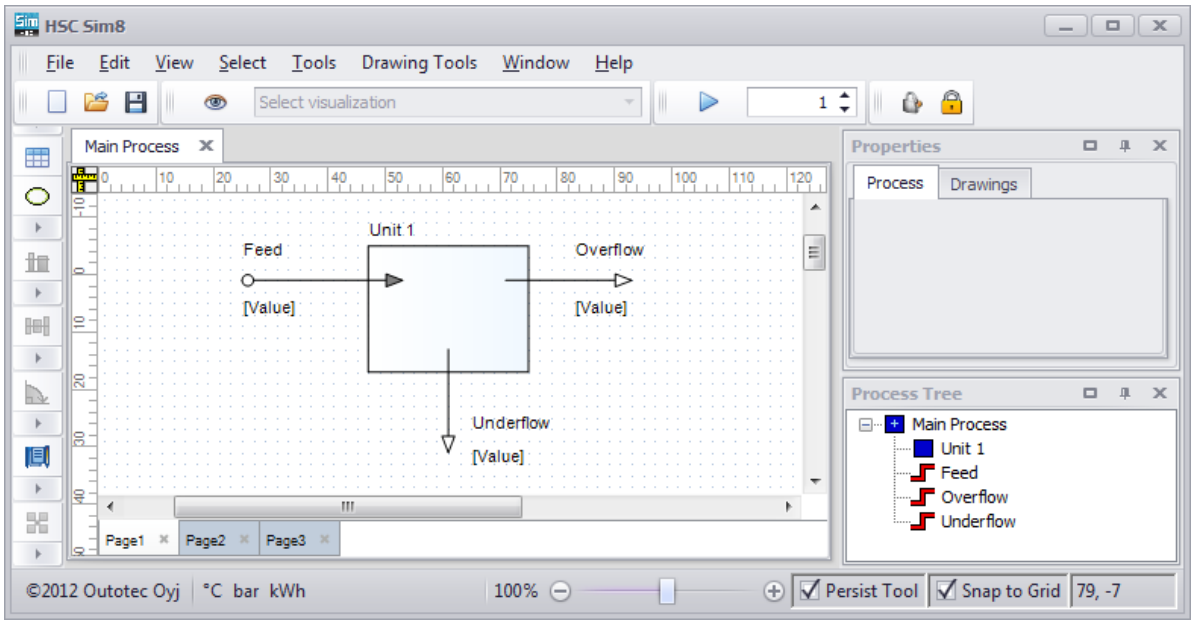


Figure 8. An example unit with renamed streams.

Double-click the unit to open its properties view. Choose Variable List Editor and add H2O to the Water Phase. Add SiO2 to Pure Phase (in the figure: Solid Phase), see Figure 9.

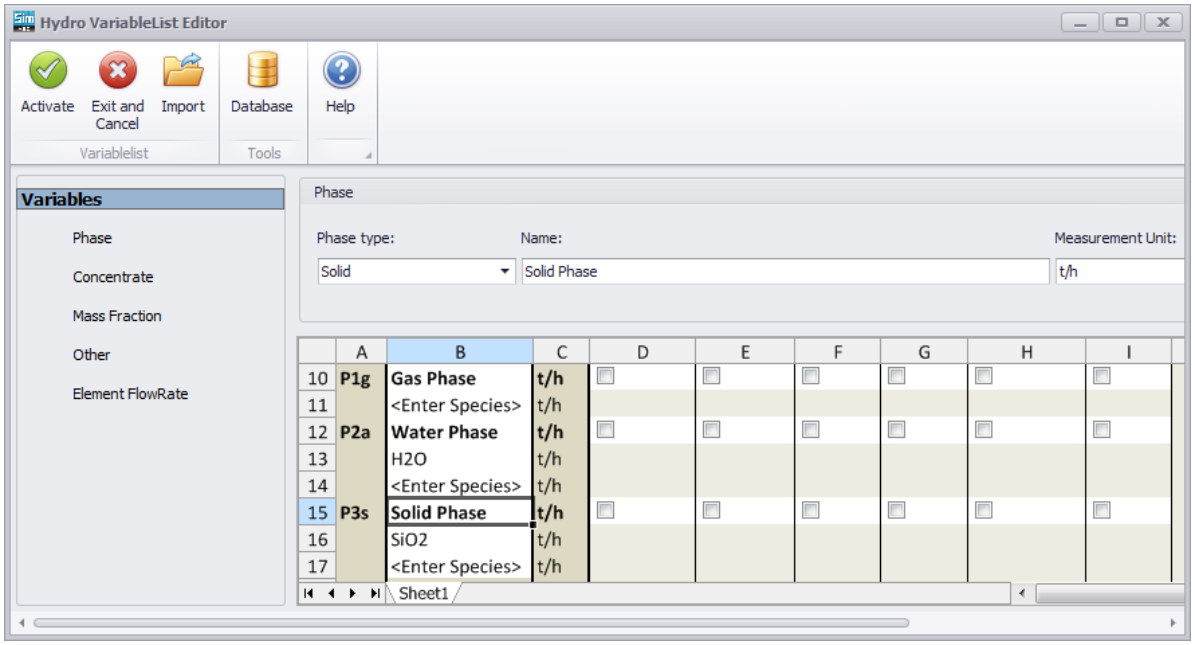


Figure 9. Variable List Editor after compounds have been added and a phase name has been modified.

Accept changes by clicking the Activate button in the top left corner. Choose the *Input* sheet and specify the amount of incoming compounds, e.g. 5 t/h of H2O and 2 t/h of SiO2. Close the properties window. Press the Simulate button (light blue right-pointing arrow) in the toolbar. After the calculations are ready, turn on the Toggle Stream Visualization Mode by clicking the eyeball-like icon in the toolbar.

For illustrative purposes “Species: H2O (t/h)” option has been selected from the dropdown menu in the toolbar, see Figure 10. The Log viewer can be made viewable by choosing View > Toolbars > Log viewer. Use the Log viewer to locate the information messages sent from the model.

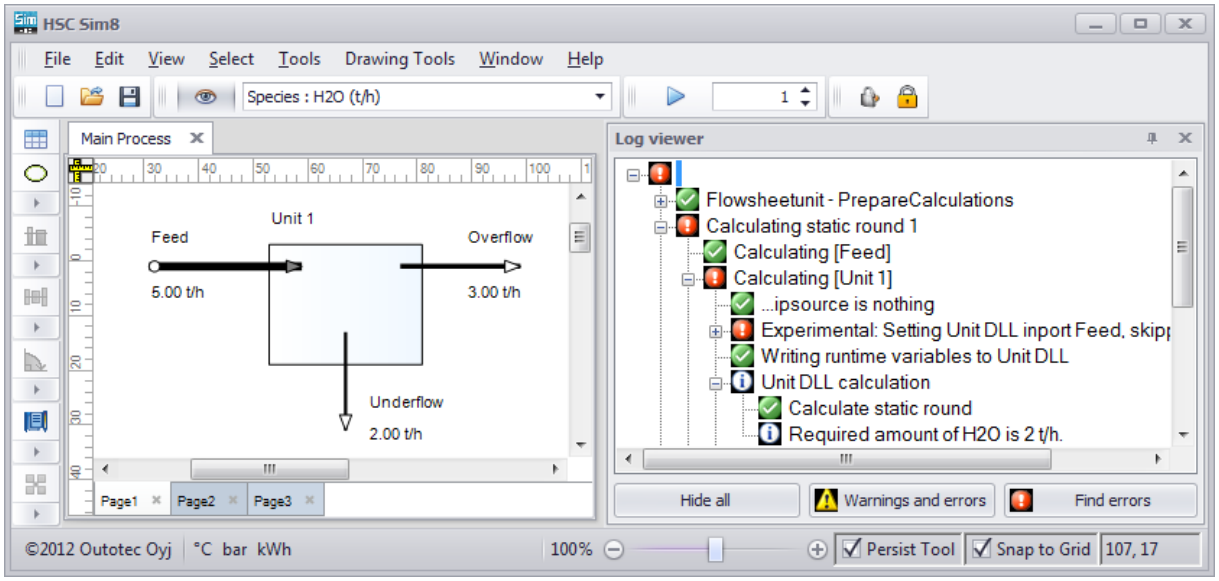


Figure 10. Streams visualized and the log viewer showing InfoMessages. Warnings can be viewed similarly through the Log viewer.

To change a parameter value, double-click the unit in the flowsheet and select the Configuration sheet. This sheet contains all user-specified parameters along with their measure units, possible limits, and descriptions of the parameters.

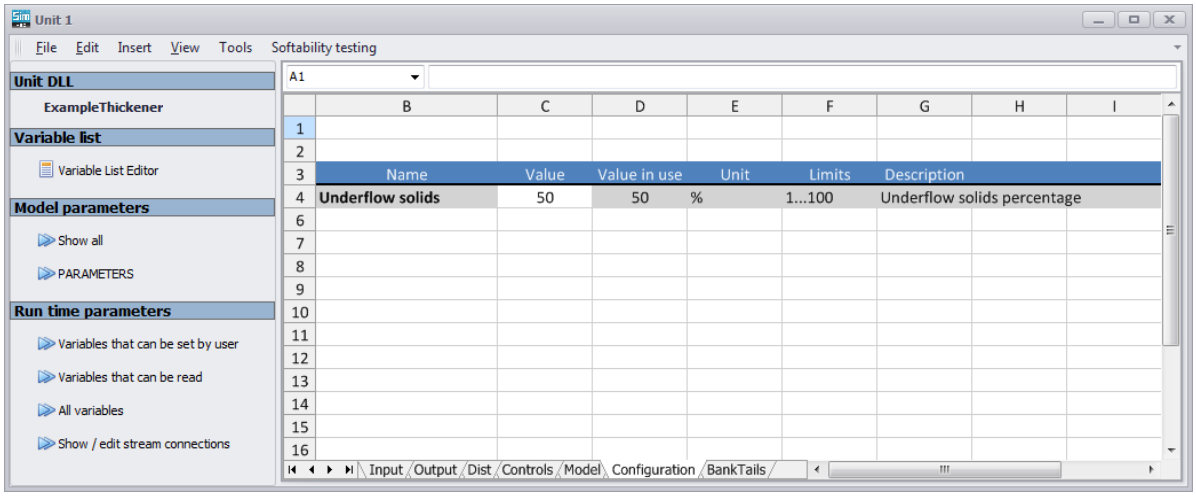


Figure 11. Users can change parameter values via the Configuration sheet.

If a value below the minimum limit is typed into the value cell, the value in the cell is automatically changed to the minimum value. A similar automatic change takes place when a value that is too great is typed into the value cell.

52.13. Appendices

Source code for the example unit

```
Imports System.Collections.Generic
Imports System.ComponentModel.Composition
Imports Outotec.HSC.Common
Imports Outotec.HSC.HSCStream
Imports Outotec.HSC.HSCExternalUnitDLL

<HSCUnitDLL("Example Unit",
    "Unit modelling example",
    "Company Name - Your Name",
    HSCUnitDLLAttribute.Mode.StaticOnly,
    TypeCode:="HU-001-01"), Export("UnitDLL", GetType(IUnitDLL8)), ExportMetadata("ClassName",
GetType(ExampleThickener))>
Public Class ExampleThickener
    Inherits UnitDLL

#Region "Stream Configuration"

    'Inputs:

    <StreamIn("Feed", 1, IsDefaultInput:=True)>
    Public Property Feed As IHSC8Stream

    'Outputs:

    <StreamOut("Overflow", 1, Required:=False)>
    Public Property Overflow As IHSC8Stream

    <StreamOut("Underflow", 2)>
    Public Property Underflow As IHSC8Stream

#End Region

#Region "Model Parameters"

    Const NODE0 As String = "PARAMETERS"

    <Parameter(NODE0, "Underflow solids", "%", "Underflow solids percentage",
               MinLim:=1, MaxLim:=100)>
    Public Property UnderFlowSolidsPercentage As Double = 50.0

#End Region

#Region "Model Internal States"
    'Not in use in this unit
#End Region

#Region "Runtime Variables"
    'Not in use in this unit
#End Region

#Region "Static and Dynamic Calculation"

    Protected Overrides Sub CalcStatic(ByVal HSC As IHSC8)
        CalcOutput()
    End Sub

    Protected Overrides Sub CalcDynamic(ByVal HSC As IHSC8,
        ByVal DynamicInfo As IDynamicCalculationInfo)
        CalcOutput()
    End Sub
```

```
#End Region

#Region "Internal Procedures"

    Private Sub CalcOutput()

        Dim massOfSolids As Double
        Dim massOfWater As Double
        Dim requiredWater As Double
        Dim waterToOverflow As Double
        Dim x As Double = UnderFlowSolidsPercentage

        'calculate the amount of material in solid and water phases
        For Each c_amount In Feed.AllStreamComponentAmounts

            If c_amount.PhaseGroup.Phase = "Solid" Then
                massOfSolids += c_amount.Amount
            End If

            If c_amount.PhaseGroup.Phase = "Liquid" Then
                massOfWater += c_amount.Amount
            End If

        Next

        'calculate the required amount of water
        requiredWater = (100 - x) / x * massOfSolids
        waterToOverflow = massOfWater - requiredWater

        'an example of an InfoMessage
        InfoMessage("Required amount of H2O is " & CStr(requiredWater) & " t/h.")

        If waterToOverflow < 0 Then
            WarningMessage("Not enough water to proceed.")
        Else

            'transfer phases to the right streams
            For Each co In Feed.AllStreamComponentAmounts

                If co.PhaseGroup.Phase = "Solid" Then
                    Underflow.AddStreamComponentAmount(co.Component, co.PhaseGroup,
                                                       co.Amount)
                End If

                If co.PhaseGroup.Phase = "Liquid" Then
                    Underflow.AddStreamComponentAmount(co.Component, co.PhaseGroup,
                                                       requiredWater)
                    Overflow.AddStreamComponentAmount(co.Component, co.PhaseGroup,
                                                      waterToOverflow)
                End If

            Next

        End If

    End Sub

#End Region

End Class
```

```
Imports System.Collections.Generic
Imports System.AddIn
Imports System.AddIn.Pipeline
Imports Outotec.HSC.Addins
Imports System.Math

<AddIn("Name")>
<QualificationData("Static", "True")>
<QualificationData("Dynamic", "False")>
<QualificationData("TypeCode", "MU-100-10")>
<QualificationData("Technology", "Others")>
<QualificationData("Version", "1.0")>
<QualificationData("Description", "Description")>
<QualificationData("Author", "© Outotec (Finland), Authors: X")>
Public Class _Template
    Inherits UnitDLL

#Region "Stream Configuration"

    '---Inputs:

    <StreamIn("Input", 1)>
    Public Property Input As IHSC8Stream

    '---Outputs:

    <StreamOut("Output", 1)>
    Public Property Output As IHSC8Stream

#End Region

#Region "Model Parameters"
    'Not existing for this unit.
#End Region

#Region "Model Internal States"
    'Not existing for this unit.
#End Region

#Region "Runtime Variables"
    'Not existing for this unit.
#End Region

    Protected Overrides Function HiddenPropertyNames() As HashSet(Of String)
        Dim lst = New HashSet(Of String)


        Return lst

    End Function

    Protected Overrides Sub InitModelParameters(ByVal HSC As IHSC, ReferenceFeed As IHSC8Stream)

    End Sub

#Region "Static and Dynamic Calculation"

    Protected Overrides Sub CalcStatic(ByVal HSC As IHSC)

    End Sub

    Protected Overrides Sub CalcDynamic(ByVal HSC As IHSC, ByVal timestepsseconds As Integer)

    End Sub

#End Region

#Region "Internal Procedures"


#End Region

End Class
```

Figure C.1 Template for the HSC Chemistry Sim unit operations.

```
Imports System.Collections.Generic
Imports System.AddIn
Imports System.AddIn.Pipeline
Imports Outotec.HSC.Addins

<AddIn("Perfect Mixer")>
<QualificationData("Static", "True")> <QualificationData("Dynamic", "True")>
<QualificationData("TypeCode", "MU-110-10")> <QualificationData("Technology", "Concentrator - General")>
<QualificationData("Version", "1.0")>
<QualificationData("Description", "Mixes all input material from one or several streams and passes it equally
to one or several outputs.")> <QualificationData("Author", "© Outotec (Finland), Authors: Antti Remes")>
Public Class PerfectMixer
    Inherits UnitDLL

#Region "Stream Configuration"

    <StreamIn("Input", 1, IsDefaultInput:=True)> 'Inputs
    Public Property Input As IHSC8Stream

    <StreamOut("Outputs", 1)> 'Outputs
    Public Property AllOutputs As List(Of IHSC8Stream)

#End Region

#Region "Model Parameters" 'Not existing for this unit.
#End Region
#Region "Model Internal States" 'Not existing for this unit.
#End Region
#Region "Runtime Variables" 'Not existing for this unit.
#End Region

#Region "Static and Dynamic Calculation"

    Protected Overloads Overrides Sub CalcDynamic(HSC As IHSC, timestepseconds As Integer)
        CalculateOutput()
    End Sub

    Protected Overloads Overrides Sub CalcStatic(HSC As IHSC)
        CalculateOutput()
    End Sub

#End Region

#Region "Internal Procedures"

    Private Sub CalculateOutput()

        Dim outCount As Integer = AllOutputs.Count ' Number of outputs
        Dim useFastMethod = True

        If Not useFastMethod Then
            ' This is the basic AddStreamComponentAmount loop which is here as an example. However, it is
            ' slightly slower than the AddStream function call
            ' so it is not used in the production code

            Dim i As Integer
            For Each st In AllOutputs
                For Each ca In Input.AllStreamComponentAmounts
                    AllOutputs(i).AddStreamComponentAmount(ca.Component, ca.PhaseGroup, ca.Amount / outCount)
                Next
                i = i + 1
            Next
        Else
            ' A faster way to distribute streams evenly (just a single AddStream call for every output)
            For Each st In AllOutputs
                st.AddStream(Input, 1.0 / outCount)
            Next
        End If

    End Sub

#End Region

End Class
```

Figure D.1 Code for a sample HSC Chemistry Sim unit operation, the Perfect Mixer.

Table E.1 Heuristics for interaction design as expressed by Nielsen (1995).

General principle
Visibility of system status The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.
Match between system and the real world The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
User control and freedom Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
Consistency and standards Users should not have to wonder whether different words, situations, or actions mean the same thing.
Error prevention Even better than good error messages is a careful design, which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.
Recognition rather than recall Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
Flexibility and efficiency of use Accelerators – unseen by the novice user – may often speed up the interaction for the expert user so that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.
Aesthetic and minimalistic design Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in dialogue competes with the relevant units of information and diminishes their relative visibility.
Help user recognize, diagnose, and recover from errors Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
Help and documentation Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.